12-1987

The Western Michigan University Programming Contest Proctor System

Fletcher

Follow this and additional works at: https://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation

https://scholarworks.wmich.edu/masters_theses/1220
THE WESTERN MICHIGAN UNIVERSITY PROGRAMMING
CONTEST PROCTOR SYSTEM

by

Joel Marshall Fletcher

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
December 1987
A software package for administering programming contests has been developed for DECsystem-10 timesharing computers. This package provides a neutral editing environment for contestants, a method for submitting problems for execution, and an automatic judging facility for evaluating contestants' solutions and tabulating scores. The proctor system has been used in numerous competitions, including a regional contest sponsored by the Association for Computing Machinery.
ACKNOWLEDGEMENTS

A project of this magnitude can rarely be completed in a vacuum. Many people have provided helpful contributions to the development of the Proctor System, and to the paper describing this software.

First, I extend my thanks to my advisor, Dr. Mark Kerstetter, whose encouragement and guidance have enabled me to bring this project to its present end.

To my parents and my wife Katherine, I can never offer enough thank-yous. Their patience and support have approached the infinite.

My thanks also go to Ronald Schubot, who co-authored the first Proctor System, and provided much input and assistance when I needed another hand, or another "Good Idea."

Many others have helped the project along the way, including the Computer Club of WMU, the Academic Computer Center, the Computer Science department, Mrs. Caroline Sleep, and the many participants in local and regional programming competitions. Their contributions have been greatly appreciated.

Finally, I would like to pay a special rememberance to Jim Sleep, who provided the initial inspiration for this project, and who encouraged my enormous interest in computing, as he did so for so many others.

Joel Marshall Fletcher
INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International
300 N. Zeeb Road
Ann Arbor, MI 48106

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The Western Michigan University programming contest proctor system

Fletcher, Joel Marshall, M.S.
Western Michigan University, 1987

Copyright ©1987 by Fletcher, Joel Marshall. All rights reserved.
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs or pages
2. Colored illustrations, paper or print
3. Photographs with dark background
4. Illustrations are poor copy
5. Pages with black marks, not original copy ✓
6. Print shows through as there is text on both sides of page
7. Indistinct, broken or small print on several pages
8. Print exceeds margin requirements
9. Tightly bound copy with print lost in spine
10. Computer printout pages with indistinct print
11. Page(s) lacking when material received, and not available from school or author.
12. Page(s) seem to be missing in numbering only as text follows.
13. Two pages numbered. Text follows.
14. Curling and wrinkled pages
15. Dissertation contains pages with print at a slant, filmed as received
16. Other

________________________________________________________
________________________________________________________

UMI

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ........................................... ii
LIST OF TABLES ............................................... v
LIST OF FIGURES ............................................ vi
INTRODUCTION ...................................................... 1

CHAPTER

I. THE PROGRAMMING CONTEST ENVIRONMENT .................. 3
   Background: The Assessment of Programming Skills ........ 3
   The ACM Scholastic Programming Contest ..................... 4
   Summary .................................................. 8

II. GOALS OF THE DESIRED SYSTEM ............................ 10
   General Structure ........................................... 10
   User Interface Considerations ........................................... 12
   Contest Administrator ........................................ 15
   Judging Process ............................................ 16
   System-wide Design Issues ................................... 17

III. THE WESTERN MICHIGAN UNIVERSITY PROGRAMMING CONTEST PROCTOR SYSTEM .................. 21
   Background .................................................. 21
   The Western Michigan University Computing Environment .... 22
   Overview of System Components ............................... 23
Table of Contents--Continued

Additional Proctor System Utilities .................. 32
Design and Implementation Methodology ............ 37

IV. PROCTOR SYSTEM INTERNAL STRUCTURE ........... 39
   The Programming Contest Message Service .......... 40
   PCSINFO Memory Resident Shared Database .......... 49

V. TESTING AND USE OF THE PROCTOR SYSTEM .......... 55
   Simulation and User Testing .................... 55
   Production Use of the Proctor System ............. 56

VI. CONCLUSIONS ................................... 64
   Operational Results ............................. 65
   User Participation .............................. 66
   Future Research ................................. 66

APPENDICES ....................................... 73
   A. COPS User Guide ............................... 74
   B. Proctor System Symbol Definition File .......... 84
   C. Proctor System Contest Administrator's Reference Guide 87
   D. PCSINFO Shared Data Segment With Access Routines .... 93

BIBLIOGRAPHY ..................................... 97
## LIST OF TABLES

1. COPS Command Definitions ........................................ 26
2. JUDGE Command Definitions ..................................... 33
3. PCS Message Service Classes ................................... 40
4. PCS Shared Database Description ............................... 53
LIST OF FIGURES

1. Batch Mode Contest Structure .................  10
2. Proctor System Components ...................  24
3. COPS Message State Diagram for Problem (p) .......  42
4. Message Retry Algorithm .......................  47
5. Proctor System memory Organization .............  51
6. Job and CPU utilization During Nov. 18 contest ....  58
7. Disk and Swapping Loads During Nov. 18 contest ....  58
8. Microcomputer-Based Contest Organization ..........  68
9. Network-Based Contest Organization ..............  70

vi
INTRODUCTION

The effort to be described in this paper is the design, development, and implementation of the Western Michigan University Programming Contest Proctor System, a software package to provide a computing environment suitable for hosting intercollegiate programming competitions such as those sponsored by the Association for Computing Machinery (ACM).

The thesis first describes the ACM contest itself and its value as a tool for evaluating programming skills. Past and current methods of administering these competitions are described next, along with the strengths and weaknesses of each method. Following is a discussion of the goals for an improved computing environment for holding ACM-style programming contests. The WMU Programming Contest Proctor System is then presented, as implemented on the DECsystem-10 timesharing computer.

Chapter four presents the internal structure of the Proctor System with an emphasis on two particular aspects of the software architecture which establish the framework of the system. An inter-process communications facility has been devised to implement a message passing system for use between system components. In addition, an unusual memory organization has been adopted to provide a memory resident database which is shared by all Proctor System programs.
The succeeding sections of the thesis discuss the application of the Programming Contest Proctor System in the 1984 ACM East Central Regional competition, where forty-seven teams made use of the proctoring software. The performance of the system is analyzed, and guidelines are established for running the Proctor System on various hardware configurations.

The paper concludes with an evaluation of the success of the Proctor System project, based on the experience gained with the Proctor System software in local and regional competitions. The personnel requirements for managing the regional competition using the Proctor System are outlined as well. Finally, conclusions are drawn about the adaptation of the Proctor System model to alternate hardware and software environments, and the applicability of the Proctor System to classroom use.
CHAPTER I

THE PROGRAMMING CONTEST ENVIRONMENT

Background: The Assessment of Programming Skills

A significant challenge confronting computer science educators is the evaluation of students' programming and problem solving abilities. It is of particular importance to accurately assess programming aptitude in introductory courses where instruction centers on developing programming skills, rather than presenting advanced theory. In university environments, the evaluation process is often divided between classroom examinations and take-home programming assignments. While each method provides some measure of students' progress, each has limited effectiveness in assessing programming ability.

Classroom examinations provide some measure of the student's understanding of the concepts presented in class. However, they often fail to test the student's ability to use the concepts effectively. This occurs because handwritten solutions to sufficiently complex programming problems are difficult to grade.

Programming problems that are given as homework assignments have the needed complexity to challenge students' abilities, but are not easily controlled. That is, the instructor has no way of determining the time or machine resources spent by any given student on a problem, or what assistance the student may have received.
Lorenzen\(^1\) points out that while the level of assistance shared among students is admirable, "sometimes it results in a student handing in a program that the student could not have produced alone."

It is clear that in many circumstances, the ideal method of assessing programming skills is to present students with problems to be solved on actual computing hardware, but with time restrictions imposed on the student, and some form of proctor to ensure each student works independently. These concepts are embodied in the ACM Scholastic Programming Contest.

The ACM Scholastic Programming Contest

**Introduction**

In 1975, the Association for Computing Machinery (ACM), a society for computer science professionals, began sponsoring annual intercollegiate programming contests. These competitions, featuring teams from colleges and universities worldwide, are held in 11 North American regions and one international region. The top two teams from each region then advance to a final competition, which is held in conjunction with the annual ACM Computer Science Conference.

**Description of the Contest**

The ACM Programming Contest is a problem solving competition. Each four-student team is assigned a set of four to six problems to

---

solve in a fixed time, normally four hours. All teams are given identical problems to solve, and each team must write a computer program to solve each problem. Standard sets of input data are created by the judges to test the correctness of the proposed solutions to each problem. Problems may be solved using either of the programming languages FORTRAN or Pascal.

A score is computed for each problem that has been successfully programmed. While exact scoring procedures vary from contest to contest, the following scoring model is typical of those found in many competitions:

1. For each correctly programmed solution, one penalty point is assessed for each minute spent to solve the problem.

2. Each attempted execution of the problem with the contest data set is assessed an additional fifteen penalty points.

3. Participants are also allowed to execute their programs with practice data sets they create themselves. In many competitions, these practice runs are assessed ten penalty points each.

4. The total score for each problem is computed as the sum of the accumulated penalty points (time plus execution points).

5. Final standings are computed based on number of problems solved. Those teams with the most correct solutions place highest in the competition. When two or more teams have solved the same number of problems, those teams are ranked by score, with the team with the fewest penalty points placing highest within that group.

The scoring method employed by the ACM contest is intended to
reliably gauge problem solving ability as a combination of programming accuracy plus debugging skill and speed.

Methods of Implementation

Historically, ACM programming contests were held using batch-oriented mainframe computers. Each team had the use of one keypunch machine to punch its solution programs and practice data. The contest administrator added necessary job control cards and contest data decks to execute each program. Output listings were then examined by judges and compared to an answer key for correctness. Scores were tabulated manually, and results were posted at the conclusion of the competition.

This approach had several drawbacks. First, it required great amounts of human intervention to assemble card decks and to evaluate output data for correctness. Additionally, the manual scoring methods employed were subject to human error. Further, the serial nature of batch-oriented systems made poor use of computing resources and caused great delays in program execution. Finally, keypunches such as those used in batch-oriented contests, provided no editing capability, thus placing undue emphasis on contestants' typing accuracy, rather than their programming ability.

In recent years, several attempts have been made to conduct programming contests using microcomputers or timesharing mainframe computers, reflecting the shift throughout the computing industry to more interactive forms of computing. Interactive programming contests
are generally held utilizing the standard command language, text editor, and language processors for the competition's host computer. Contestants are taught the necessary commands to create and edit their program and practice data files, submit contest runs, and obtain printouts of their work. As in the batch oriented designs, contest runs are examined by the contest judges for correctness and points are totaled for each team.

Interactive contest structures solve many of the inherent deficiencies of batch-oriented contests. First, even the most primitive of text editors is better suited to program and data entry than the type of keypunches common to early programming contests. Interactive systems also make more efficient use of computing resources than batch environments, since the central processor need not stand idle between jobs. The elimination of card decks also alleviates some of the delays and errors associated with the handling of punched cards.

While interactive contest schemes improve considerably on earlier batch oriented contests, there are several shortcomings in such systems. Like the earlier batch systems, there is too much manual intervention in most interactive contest designs, causing great delays in program verification.

An additional deficiency of the interactive contest model is the inherent "openness" of the system; that is, the ability for each contestant to access the complete command set of the contest's host computer. While those contestants familiar with the host operating
environment are able to make effective use of available editors, debuggers, and other facilities, those who are new to the system may not be able to use the computer to their best advantage. This leads to a sort of "home team advantage" for those contestants familiar with the operating environment, often critically handicapping teams lacking knowledge of subtle system aids. The home team advantage, like the keypunches of batch-oriented contests, diminishes the value of programming skills relative to external factors; in this case, familiarity with the operating environment.

A further consequence of an interactive environment is the threat of a malicious user tampering with contestants' programs, or otherwise altering the outcome of the contest. Clearly, a timesharing system is more vulnerable to intrusion than a batch operation.

Summary

The ACM Scholastic Programming Contest provides the means to judge the relative abilities of students at solving various types of problems using digital computers. However, past and present contest implementations have not adequately addressed concerns of functionality, security, and fairness to contestants. It is with these concerns in mind that a project was initiated to develop a software package specifically tailored to manage an ACM-style programming contest environment.

It was anticipated that further investigations, aided by experience gained using a programming contest proctor facility, would
give some insight into the value of proctoring packages in a more general setting, speaking to the needs of a larger academic community. The ACM-style contest provides a well-specified environment from which a better understanding of proctoring software can be obtained.
CHAPTER II

GOALS OF THE DESIRED SYSTEM

General Structure

In order to establish the specifications for a proctoring system, it is necessary to introduce the principal elements of the contest, and describe their interactions. These components, shown in Figure 1, are the contestant, the contest administrator, the judge, and the execution batch processor. In the traditional batch-style competition, the contestant, judge, and contest administrator are humans, while the execution batch processor is a software interface to allow program execution on the central processing unit (CPU).

![Figure 1. Batch Mode Contest Structure](image-url)
In fact, Figure 1 is an accurate representation of the interactions found in the "classic" batch mode contest. From the contestant's point of view, his solution program is treated by the system in the following manner:

1. The contestant submits his solution program to the contest administrator.
2. The contest administrator runs the proposed solution through the computer's batch processing system.
3. The results of the contestant's program are retrieved by the contest administrator.
4. The results are passed on to the judge for evaluation.
5. The judge informs the contest administrator whether the contestant's solution was correct.
6. If the contestant's solution is judged incorrect, his program is returned for correction.

The paths in Figure 1 are labeled to correspond with the above process, and represent the paths taken to execute and evaluate a contestant's solution program. In an interactive programming contest model, each element in Figure 1 represents a process or task, with all tasks executing concurrently under the control of a timeshared operating system. The batch processor task is provided by the operating system to allow the submission of programs for execution in batch mode.

From this model the major functional parts of an interactive proctoring system can be specified. Three distinct parts comprise the
interactive system: the user interface, the judge's interface, and the contest administration module. The major concerns of each component must be examined individually, and certain goals must be established for the proctor system as a whole.

User Interface Considerations

The user interface portion of the proctoring system model is that part of the system which is accessed by each contestant. Its principal functions are to permit program entry, and submission of programs to the contest administrator for execution and evaluation.

In the design of a user interface, the most critical issue confronting the designer is that of "user-friendliness." In particular, it is necessary to provide a good balance between the power afforded by the command language, its ease of use, and its ease of learning. Embley and Nagy¹ note that in command languages, ease of use and ease of learning are "different and often conflicting goals."

An example of such a conflict between ease of use and ease of learning is the UNIX¹,² operating command language, the shell. Commands to the shell consist mostly of terse, two or three character command names with single character command options. Multiple

³UNIX is a trademark of AT&T Bell Laboratories.
commands may be linked on a single line, forming complex operations with a minimum number of keystrokes. While this conciseness aids the experienced user in quickly performing complex operations, Hardy1 concludes that the "terseness of the commands make them somewhat difficult to learn."

Embley and Nagy2 cite the "poor choice of command names and capricious command abbreviations" as a major shortcoming of most editors and command interpreters, along with the following:

1. Context dependent commands and multiple modes or states.
2. Too many commands and too many parameters per command.
3. The absence of a HELP feature.

Allen's3 analysis of human patterns of short-term and long-term memory confirm this notion, and further conclude that consistency in command syntax is the principal factor contributing to the user's ability to learn and recall commands.

Embley and Nagy's programming system, SIMPLE, was developed with these thoughts in mind. SIMPLE was designed as a program development environment for beginning computer science students. As a result, there are only fifteen commands in the SIMPLE command set.

---

The users of the programming contest environment are much like the users of SIMPLE, in that they are new to the programming environment of the proctoring software. Typically, contestants have no more than a few hours to familiarize themselves with the contest facilities prior to the competition itself. However, since the contestants are assumed to have prior experience in using computers, certain knowledge can be assumed, such as keyboard usage.

Accordingly, the contestants must be presented with an easy-to-learn command language and editor which provides all the functions necessary for program development, while being very easy to learn for those with prior computer experience. This requires a concise command set, commands which are easily remembered, and an online HELP facility.

The Home Team Advantage

A concise, easily learned command language can aid contestants in competing, with a minimum of previous exposure to the system. Additionally, a simple command language can serve to minimize the so-called home team advantage by shielding the contestants from the host operating system's command language interpreter. This requires that the command language be a closed system which does not allow contestants direct access to the operating system facilities. A closed system is desirable both to both shelter the contestants from the idiosyncrasies of the operating system, and to prevent knowledgeable users from taking advantage of facilities unknown to
other competitors.

Contest Administrator

The contest administrator plays a central role in the contest model outlined previously. All transactions between the contestant, the batch system, and the judge are conducted through the contest administrator, and each transaction request received by the contest administrator is passed to another module. For example, when a contestant submits a program to the contest administrator, the contest administrator passes the program to the batch system for execution.

Although it is responsible for relaying transaction requests between the other system components, the contest administration module cannot be viewed simply as a message switching mechanism. The process of submitting, executing, and evaluating a contestant's run must be viewed as a series of actions which must be executed in their proper sequence. One cannot, for example, judge a program which has not been submitted. Nor can a contestant be allowed to resubmit a program until it has passed through the entire submission-execution-evaluation process. It is necessary then for the contest administration module to maintain the state of each contestant's run, and to filter out to those messages that are not legal for a particular state.

Thus, the contest administrator process must be viewed as a large finite state machine which is able to receive messages from other modules, validate these messages, and initiate actions based on the
current state of the contestant's process and the type of message received.

Judging Process

The primary responsibility of the judge of a programming contest is the evaluation of the contestants' solution programs for correctness. While this seems a simple task, it is the judging process which has been responsible for more delays and more errors in contest results than any other factor. This is due to the fact that traditionally, all evaluation of contest runs has been performed manually.

In order to remove this deficiency, it is necessary to partially or fully automate the program evaluation process. Although it would be desirable to fully automate program judging, there are certain dangers involved in doing so. For example, it may be possible to have two correct solutions to a problem, when only one solution was anticipated by the judge. A fully automatic judging process could not detect such an event and bring it to the attention of the contest judge. Therefore, the judging facilities must operate automatically, but there must be provisions for manual evaluation of contestants' runs, in order to minimize the possibility of a judging error.
System-wide Design Issues

Security Concerns

The concern for security in a programming contest proctoring environment is similar to such concerns in any multi-user programming environment. Primarily, this centers on three areas: protecting the contestants against intrusion by other contestants, protecting all contest components and data from outside intrusion, and protecting contestants from their own mistakes.

The first area of concern involves preventing one contestant from either accidentally or maliciously accessing or altering another's data, or otherwise interfering with his activities. This is easily managed by keeping users within a "closed" programming environment, where competitors are only given access to a small, well defined set of operations. By maintaining a closed system, the most severe shortcoming of traditional interactive contest environments is eliminated. Contestants can only access the data to which they are authorized.

It is more difficult to protect contestants from malicious tampering from outside the context of the programming contest system. First, the contestants' programs and data must be protected from unauthorized access. Next, the proctor system itself must be impervious to external tampering. Finally, the operating system itself must be protected from any intrusion that would prove harmful to the contest. Often, it is not possible to make up for security...
deficiencies within the operating system, except by restricting access to the operating system to contest-related processes only, and by confining contestants' access to the limited environment of the proctor system itself.

Despite the concerns over outside tampering with contestants' data, often a novice user will be his own worst enemy at a computer terminal. Given the contestants' relative unfamiliarity with the programming contest environment, this issue is of particular concern.

A common characteristic of many operating systems is the use of file name "wild-card" schemes which allow for easy manipulation—including deletion—of large groups of files. Many text editors similarly allow for the easy deletion of large amounts of text. Under normal circumstances, a computer user will occasionally use one of these features incorrectly, resulting in the loss of some or all of his data. With the increased pressure of the programming contest environment, there is an even greater chance that he will destroy his data due to a typographical error or other misuse of wild-card features.

Therefore, a proctor system's contestant interface should make it difficult for a user to destroy large amounts of data. This can best be accomplished by eliminating wild-card capabilities, and by asking for confirmation by the user before deleting large amounts of data.

Reliability and Error Recovery

Although thorough software testing and simulation tend to
minimize the errors in a software package, the ability to recover from errors that do occur is critical. For a contest proctor system, it is particularly important to minimize the effect of software and hardware failures and to quickly recover from those that do occur, since the contest itself is on a strict timetable of a few hours.

Errors can be categorized into three major classes: errors affecting a single contestant, proctor system errors affecting all users, and operating system or hardware failures. It is rarely possible for a user program to reduce the occurrence of operating system or hardware errors. It is possible, though, to minimize the number of global contest system errors by distributing as much functionality as possible to each user process, and by eliminating or minimizing the reliance on a single, complex central process.

If it becomes necessary to restart a contest after a failure of the hardware, operating system, or central contest management software, it must be done quickly, and with minimal impact to users. The proper scores and problem completion status must also be carried over to provide accurate contest results.

Portability and Related Concerns

It is always desirable to be able to transport a software package to a variety of computers and operating systems. Often this not possible, though, due to the nature of the software package. When designing the Programming Contest Proctor System, it became clear the package could not be transported easily, since it must interface...
closely with certain operating system functions.

At the time the Proctor System was developed, there was only one computer at Western Michigan University capable of supporting the workload imposed by 35-40 simultaneous users: a Digital Equipment Corporation DECsystem-10. Thus it was decided that the Proctor System should be designed to operate with maximum efficiency on the DECsystem-10, sacrificing portability of the software.

Although portability was eliminated as a factor in the design of a proctor system, the ability to maintain and augment the package was still of considerable concern. To this end, it was decided that where possible, the proctor system should be written in a high-level programming language. The primary advantages afforded by high-level languages over assembly language were the ease with which program modules could be created, debugged, and maintained. Where necessary, assembly language modules would perform functions that could not be easily implemented in a high-level language.
CHAPTER III

THE WESTERN MICHIGAN UNIVERSITY PROGRAMMING
CONTEST PROCTOR SYSTEM

Background

The Western Michigan University Programming Contest Proctor System was originally conceived in 1979, and a rudimentary system was developed the summer of 1980 for use in a local contest. This package, PCS1, provided the basic editing and job submission facilities for a one problem competition using the FORTRAN language. The prototype system lacked several facilities outlined in the design specification, though, such as automatic scoring and error recovery. While the prototype system was written to support a contest with multiple problems, this feature was not tested extensively, and was not used in PCS1.

After some experience was gained with the prototype software, a major redesign of the Programming Contest Proctor System was begun. While retaining much of the flavor of the original system, and much of the source program, the revised system implemented the original design objectives that were omitted from the prototype. This revised package has been used to conduct several local competitions, and was utilized in the 1984 ACM East Central Regional Programming Contest.
The Western Michigan University Computing Environment

Western Michigan University operates several mainframe computers, a variety of minicomputers, and several hundred microcomputers on its campus. Of these, the major academic computing is performed on a Digital Equipment Corporation DECsystem-10\(^1\) timesharing computer. The DECsystem-10 consists of dual KL-10 central processors with 12.5 megabytes of main memory, approximately two billion bytes of disk storage, and front-end terminal controllers allowing access by over 250 local and dialup terminals. The DECsystem-10 runs the TOPS-10 operating system, and supports 175 concurrent users.

TOPS-10\(^2\) (often called the Monitor) is a timeshared operating system which allows simultaneous access by interactive users and batch processes. The file system in TOPS-10 supports hierarchically structured directories, and a flexible file protection scheme which can grant file access based on user identification, group identification, or program identification.

The TOPS-10 Process Software Interrupt system (PSI) allows individual user jobs to trap asynchronous software events. Several mechanisms for inter-process communication exist within TOPS-10 to allow cooperating user jobs to exchange messages and signal software events. The Monitor supports a general purpose command language

\(^1\)DECsystem-10 is a trademark of Digital Equipment Corporation.

containing over 200 commands. A limited facility also exists for providing user written command interpreters, which are run at any time TOPS-10 would normally enter command mode.

Symmetric Multiprocessing, or SMP, is a feature of TOPS-10 which allows multiple central processors to be utilized in such a way that they appear to users as a single computing system. Further, SMP allows user programs to continue unaffected on remaining hardware should one or more processors fail. The major benefits SMP provides in the programming contest environment are increased capacity, increased throughput, and a lower likelihood of contest interruption due to hardware failure.

The combination of large capacity, a flexible programming environment, and the additional protection afforded by SMP make the DECsystem-10 well suited to the Programming Contest Proctor System.

Overview of System Components

The WMU Programming Contest Proctor System is a collection of programs which run as standard user processes under control of the TOPS-10 operating system (see Figure 2). Several major components of the Proctor System provide interfaces between the user, contest administrator, and operating system. An inter-process communications system links the various system components, and several ancillary programs serve utility functions. At the heart of the system are three programs: COPS, the Contest Oriented Programming System, PCSINFO, the Programming Contest System Information Manager, and
JUDGE, the judging and system administration facility. The Proctor System is also linked to the GALAXY-10 spooling and batch system through QUASAR, the GALAXY queue manager. The relative roles of COPS, PCSINFO, JUDGE, and QUASAR are identical to their counterparts in the classic batch model of Figure 1. Although only two COPS processes are shown in Figure 2, there may be up to 63 COPS processes in actual practice.

COPS - The Contest Oriented Programming System

COPS, the user command processor and editor, is the part of the Programming Contest System seen by the contestants themselves.

Figure 2. Proctor System Components
Each team in the competition is assigned a terminal which runs the COPS program, and allows editing of one program file and one practice data file for each problem assigned in the contest. COPS provides a user interface modeled on the Dartmouth BASIC\textsuperscript{1} environment. Since most programmers have had some experience with a BASIC interpreter at one time or another, it comes as close as possible to a universal operating environment. Consistent with the requirement that the command interface be easily mastered, COPS only has eleven commands, even fewer than Embley and Nagy's SIMPLE. Thus, even those without experience with BASIC can become comfortable with the COPS environment after a few minutes of use. Table 1 lists the entire COPS command set and the function of each command.

Contestants' program and data files are treated as standard TOPS-10 text files, with sequence numbers assigned to each line. Line insertion and replacement is accomplished by typing the line sequence number, followed by the text to insert or replace on that line.

The use of line sequence numbers presents one minor inconvenience to contestants. The Pascal and FORTRAN runtime input/output libraries cannot read files with line sequence numbers. For this reason, contestants' practice data files are written without sequence numbers, then resequenced when next edited. Thus, a data file may have different sequence numbers than it had on the previous edit. This could be corrected in a future system by writing two copies of the

\footnotesize{\textsuperscript{1}J. G. Kemeny and T. E. Kurtz, \textit{BASIC}, (Hanover NH: Dartmouth College, 1961) pp. 10-15.}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
edited data file; one with sequence numbers, and one without.

As the data files created for the contest are usually quite small, and are not edited often, the inconsistency of line numbering in this case is not seen to be a major problem. Furthermore, this problem does not occur for the FORTRAN or Pascal programs themselves, as both compilers are able to read line-sequenced program files.

Table 1
COPS Command Definitions

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>Selects data file for editing.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Deletes one or more lines of text.</td>
</tr>
<tr>
<td>HELP</td>
<td>Provides online help for specific commands.</td>
</tr>
<tr>
<td>LIST</td>
<td>Lists one or more lines of text on the user's terminal.</td>
</tr>
<tr>
<td>NEW</td>
<td>Erases the entire file to start over.</td>
</tr>
<tr>
<td>PRINT</td>
<td>Lists the selected program or data on a line printer.</td>
</tr>
<tr>
<td>PROBLEM</td>
<td>Selects specified problem number for editing.</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>Selects program file for editing rather than data.</td>
</tr>
<tr>
<td>RESEQUENCE</td>
<td>Renumbers the current file.</td>
</tr>
<tr>
<td>RUN</td>
<td>Runs the selected program with practice or contest data.</td>
</tr>
<tr>
<td>STATUS</td>
<td>Displays the current status of the team's problems.</td>
</tr>
</tbody>
</table>
Most COPS commands perform identical functions to the standard
BASIC command set, including DELETE, LIST, NEW, PRINT, and RESEQUENCE. The HELP command provides online assistance with each of the COPS commands.

Several commands were added to the BASIC command set which control the programming contest environment. These commands, PROGRAM, DATA, PROBLEM, and STATUS are used to manipulate one's editing context, and to determine the present status of the team's problems. The PROBLEM command is used to select a different contest problem for editing. The PROGRAM and DATA commands inform COPS whether the user wishes to edit his program or data file. The PROGRAM, DATA, and PROBLEM commands share the common characteristic of modifying the user's editing context in the same manner as the BASIC NEW command, but hide the underlying file names and structures from the contestant.

The RUN command operates somewhat differently from its BASIC counterpart, since a contestant may execute his program using practice data, or make a judged run with contest data. When the user types RUN, he is asked which type of run he wishes to make. A practice run is executed directly at the contestant's terminal, using the data he created while in DATA mode. A contest run issues a request for a batch job to be executed, which will test the user's program with the contest data set. Using a batch job isolates the contest run from the contestant so that the contestant cannot interfere with the execution or evaluation of the run or gain access to the contest data set.

Finally, the STATUS command informs the user of his team's
current status; which problem is currently being edited, and which problems are completed or awaiting completion of a contest run.

Further examples of the COPS command language can be found in Appendix A.

PCSINFO - The Proctor System Information Manager

The central component of the WMU Programming Contest Proctor System is PCSINFO, the Programming Contest System Information Manager. The primary functions of PCSINFO are providing access validation for contestant processes, maintaining the online automatic scoring for the entire contest, and initiating contest runs via the GALAXY-10 spooling and batch system. By communicating with other contest system processes, PCSINFO is able to maintain a database containing the status of all teams. This database is accessible to other system components.

The first function required of PCSINFO is to provide a signon facility which controls access to the Proctor System for COPS processes. Each team in the programming contest is assigned a terminal to use, and is logged into the TOPS-10 operating system by the contest administrator under a unique user identification or project-programmer number. When first starting, each COPS process signs onto PCSINFO by sending an identification message to PCSINFO, giving the process number and user identification of the sender. If no other process with that identification is currently using the contest system, PCSINFO grants access to the contest. Otherwise, the
user is denied access to the contest and logged off the DECsystem-10.

Once a COPS process has signed onto PCSINFO, and has been granted access to the contest, it is the responsibility of PCSINFO to accumulate the count of practice runs and contest runs submitted by that process. An inter-process message facility allows COPS processes to signal these events to PCSINFO for tabulation. These data are maintained by PCSINFO in shared memory for use by other components of the programming contest system.

The final responsibility of PCSINFO is the submission of contest runs to the GALAXY-10 batch system. When a COPS process informs PCSINFO that the contestant wishes to execute his program using contest data, a batch job is submitted which executes his program and evaluates the output data for correctness. This function was assigned to PCSINFO to ensure that contest data are not accessible to contestants or outside users.

JUDGE - The Judging and System Administration Utility

The judging and system administration module, JUDGE, provides the interface between the contest administrator and the PCSINFO program. In addition, JUDGE is utilized by the automatic scoring facility to record correct and incorrect execution of problems using contest data.

For the programming contest administrator, JUDGE serves as the command interface to the Programming Contest Proctor System. Through the JUDGE program, the administrator controls several facets of contest operation, including setting the starting and finishing time.
of the contest, broadcasting messages to contestants' terminals, and generating batch scripts for each problem to be judged. The principal function of JUDGE, however, is the recording of the results of contest runs. This may be done by the contest administrator manually, or automatically when executed from a contest-run batch script.

Contest Run Execution and Scoring

At this juncture it is useful to discuss how contestants' problems are executed with contest data. Unlike practice runs, which are executed with the users' own data on their own terminals, contest runs are executed by batch jobs initiated by the contest controller, PCSINFO. By executing contest runs under batch control, contestants are prevented from viewing the contest data or from interfering with the execution and evaluation of their programs.

Prior to the start of the programming contest, the contest administrator must use the JUDGE facility to generate batch control scripts for contest runs. It is necessary to generate a unique command file for each problem to be solved by each team, since the GALAXY-10 batch system does not allow parameters such as team and problem numbers to be passed to batch jobs.

When a contestant requests that his program be run with contest data, the appropriate batch script is selected by PCSINFO and submitted to the batch controller for execution.

Each batch script is divided into four phases: compilation, module linking and loading, program execution, and output
verification. At each of the first three steps, any error will trigger the JUDGE program, which will inform both PCSINFO and the contestant's terminal of the error in execution. The error will also be printed in the batch log file which is printed on the DECsystem-10 line printer, immediately after the contest run is terminated.

In the verification phase of the contest run, the results produced by the contestant's program must be validated against a standard output key. This comparison may be done by either of two programs, selectable by the contest administrator. The standard DECsystem-10 file comparison utility, FILCOM, is useful where an exact character for character match between the contestant's output and the standard key is required. An alternate program called DIFF was created to compare data that are in an unstructured form; particularly numeric data where spacing is not important to the result.

If no errors are detected by the file comparison program, whether FILCOM or DIFF, the JUDGE program is invoked to inform PCSINFO and the contestant that his problem was solved correctly. If the user's program executes successfully, but the output data do not match the answer key, the JUDGE program is not invoked. Rather, a log of the batch job is printed automatically, with a list of differences between the contestant's output and the standard key. The contest administrator may inspect the list of differences and use the JUDGE program to record manually whether the contestant correctly solved the specified problem.
The decision to evaluate incorrect solutions manually stems from the belief that while automatic systems can be relied upon to judge correctness of output, they should not be trusted to determine incorrectness of the data. Since suspected differences between the contestants' results and the output key are highlighted for the judge's perusal, even manual evaluation of contest runs can be accomplished easily.

Table 2 shows a summary of the judging and system administration functions available in the JUDGE program. The only feature of the JUDGE program used in automatic judging mode is the JUDGE command itself, which records the successful or unsuccessful completion of the contestant's contest run. The remainder of the JUDGE commands: CTL, START, FINISH, BROADCAST, CHANGE, and STATUS are used only by the contest administrator to control the operation of the programming contest. Appendix C contains a full description of the JUDGE program.

Additional Proctor System Utilities

Several auxiliary computer programs have been developed to support the Programming Contest Proctor System. These programs, though less visible than COPS, PCSINFO, and JUDGE, provide essential services within the programming contest environment. The six auxiliary modules are PCSEED, PCSFSS, PCSDPY, PCSCOR, PCSDMP, and RUNCLI.
Table 2
JUDGE Command Definitions

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROADCAST</td>
<td>Sends a common text message to all contestants.</td>
</tr>
<tr>
<td>CHANGE</td>
<td>Modifies items in shared database.</td>
</tr>
<tr>
<td>CTL</td>
<td>Generates batch control files for contest runs.</td>
</tr>
<tr>
<td>FINISH</td>
<td>Sets the contest finish time.</td>
</tr>
<tr>
<td>JUDGE</td>
<td>Records successful or unsuccessful completion of a contest run.</td>
</tr>
<tr>
<td>START</td>
<td>Sets the contest start time.</td>
</tr>
<tr>
<td>STATUS</td>
<td>Displays status of all unprocessed contest runs.</td>
</tr>
</tbody>
</table>

Automatic Login of Contestant Jobs: PCSEED

For each team competing under the Programming Contest Proctor System, there must be a timesharing job logged onto the DECsystem-10. The job must be logged in under a unique project-programmer number (user identification) and connected to a specific terminal, to which the team is assigned. Since the Proctor System supports up to 63 teams, the task of logging each team in at the beginning of a competition could take considerable time; perhaps twenty minutes or more. If a contest restart is required, such a delay in the middle of the contest can seriously affect the outcome of the competition.
PCSEED is a program designed to log the contestant jobs onto the DECsystem-10 and "seed" them to the proper terminals. The contest administrator must supply PCSEED with a file containing the team number, project-programmer number, password, and terminal number of each team. PCSEED logs each team onto its private account, then attaches to the team's terminal and starts the COPS editor. By using PCSEED to perform the login procedure automatically, a competition involving fifty teams can be started or restarted in under five minutes. An additional benefit of the PCSEED is that contestants need never be given their login password, making it impossible to gain unauthorized access to the DECsystem-10.

Maintaining Contest Scoring Information: PCSDPY, PCSCOR, and PCSDMP

The programs PCSDPY and PCSCOR are used to record the contest standings. PCSDPY is designed to run on special video terminals during the contest, providing a display of the current score for each team, along with a breakdown of which problems each team has completed. Runs which are awaiting execution or evaluation are also highlighted in this display. It is possible to have several terminals running PCSDPY simultaneously in order to provide current contest standings to contestants, judges, and observers.

PCSCOR is a program derived from PCSDPY which merely prints a report of the final contest standings. The scoresheet is ordered by the number of problems correctly solved. Within each group, scores are computed based on the number of minutes taken to complete each
problem, plus a penalty assessment for each incorrect run. A penalty assessment for each practice run may be counted as well, at the option of the contest administrator. The penalty values are determined by the contest administrator prior to the start of the contest. PCSCOR is run at the conclusion of the contest, but must be run prior to stopping PCSINFO, since scoring information is obtained from the PCSINFO memory resident database.

Another utility written for the Programming Contest Proctor System is PCSDMP, the contest dump utility. The function of PCSDMP is to display the PCSINFO memory resident database directly on a computer terminal. This program is intended for use by systems programmers to isolate errors in the shared database, such as an improperly judged contest run. Database errors can be corrected by the contest administrator by issuing the CHANGE command in the JUDGE program.

Proctor System Error Recovery: PCSFSS

Perhaps the most important component of the Programming Contest System, apart from the three principal programs is PCSFSS, the Programming Contest Failsafe System. PCSFSS is a utility which allows the contest system to restart after a failure of the operating system, or of the entire Proctor System. PCSFSS acts as a daemon process which records a snapshot of the PCSINFO memory-resident database to a disk file each minute. The snapshot file contains all the necessary information to effect a restart of the contest. If the contest must be restarted, PCSINFO will read the data recorded by PCSFSS and
restore the scoring database; providing accurate scoring information even though the contest has been interrupted.

**Providing Re-Entry into COPS: RUNCLI**

The final utility written for use in the Programming Contest Proctor System is RUNCLI, a general-purpose utility used to invoke alternate command language interpreters. Often it is desirable for an operating system to restrict the commands or programs available to various groups of users, or to provide an alternate command interpreter altogether. TOPS-10 provides a method for creating user-defined command interpreters; programs which are executed automatically when the user would otherwise enter TOPS-10 command mode. A serious restriction on this feature, however, is that such programs must be installed with special privileges by the DECsystem-10 system administrator.

RUNCLI is a program designed to make a wide variety of alternate command interpreters available to users, with fewer restrictions on their use. The program makes use of an authorization file which lists the names of all alternate command interpreter programs, along with the users who are authorized to use each program. When a user wishes to run an alternate command interpreter, TOPS-10 invokes RUNCLI, which in turn runs the appropriate command interpreter program. The DECsystem-10 system administrator can make new command interpreters available to the user community at any time, simply by adding its name and a list of the authorized users to the RUNCLI authorization file.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
In the case of the Programming Contest Proctor System, RUNCLI is run once upon LOGIN, and is thereafter invoked by TOPS-10 after each practice run of a contestant's program. RUNCLI in turn runs the COPS command interpreter and editor.

Because of its value in more general applications, RUNCLI has been incorporated into the DECsystem-10 system software library at Western Michigan University, and has been utilized in several demonstration software packages.

Design and Implementation Methodology

The principal concerns implementing the Programming Contest Proctor System centered on three areas: ease of implementation, reliability of the software, and ease of maintenance and extension. Several steps were taken to address these issues.

Consistent with good programming and software engineering practice, the Proctor System was thoroughly designed before any of the software was actually written. Particular attention was paid to the communications interfaces and shared memory structures common to the entire system. This allowed the simultaneous development of several software modules, and stepwise testing and refinement of each program unit. The well-defined communications architecture also allowed critical interfaces to be tested thoroughly by load-testing simulation software. By combining these techniques with effective software testing methods the Proctor System was implemented in a relatively short time, and has proved to be a highly reliable piece of software.
Implementation Language

Of additional benefit in the development of the Proctor System was the use of a high-level language in most modules. With the exception of PCSINFO, all programs in the proctor system were written predominantly in SAIL, a programming language derived from ALGOL-60. The SAIL language features very flexible character string operations which facilitate the development of highly interactive software. SAIL also allows programmers to utilize TOPS-10 system services more easily than many high-level languages. Finally, SAIL allows for a wide variety of memory management schemes, allowing the programmer to generate shared program or data segments.

The Proctor System programs total nearly 4100 lines of source code, of which approximately 3000 lines are written in SAIL. The PCSINFO program is written entirely in MACRO-10 assembly language, and contains about 600 lines of code. The remaining 500 lines of MACRO-10 code form a common subroutine library called by the various Proctor System programs.

The use of SAIL as the principal implementation language allowed for rapid development of the Proctor System, and eased considerably the testing and later extensions to the software.

CHAPTER IV

PROCTOR SYSTEM INTERNAL STRUCTURE

In order to provide a proctoring facility which was capable of supporting a large contest environment (originally projected at 30 to 40 teams), careful consideration was given to the internal software organization of the Programming Contest Proctor System itself. The software structure was designed to be simple and reliable, to minimize the possibility of failures during a competition.

It has been noted that the Proctor System embodies many independent processes or jobs running concurrently under the TOPS-10 operating system. It is necessary for those processes to exchange information for such functions as logging into PCSINFO, submitting contest runs, and broadcasting messages to contestants' terminals. Other requirements include the ability for processes to retrieve information about contestant status, starting and finishing times, and other data.

Two facilities were developed to meet these inter-process communications needs: the Programming Contest Message Service, and the PCSINFO memory resident shared database. These two software structures form the basis of the multi-process architecture of the Proctor System, and serve to define the structure of the entire system, and the interactions between its components.
The Programming Contest Message Service

The Programming Contest Message Service is a set of subroutines which provide a simple method for various Proctor System components to signal events to other processes. All inter-process messages issued between Proctor System components fall into six classes: Hello, PracticeRun, ContestRun, RunSuccess, RunFail, and Broadcast. The function of each message type is described in Table 3.

Figure 3 shows a simplified state diagram for the user COPS process. The state diagram represents the three states possible for each problem (p) to be solved, and the messages which can cause a state transition.

Table 3

PCS Message Service Classes

<table>
<thead>
<tr>
<th>Message</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello(team_number)</td>
<td>COPS signon message</td>
</tr>
<tr>
<td>PracticeRun(team_number,problem_number,0)</td>
<td>Practice run request</td>
</tr>
<tr>
<td>ContestRun(team_number,problem_number,0)</td>
<td>Contest run request</td>
</tr>
<tr>
<td>RunSuccess(team_number,problem_number,0)</td>
<td>Contest run success</td>
</tr>
<tr>
<td>RunFail(team_number,problem_number,error_code)</td>
<td>Contest run failure</td>
</tr>
<tr>
<td>Broadcast(broadcast_number,0,0)</td>
<td>Broadcast message</td>
</tr>
</tbody>
</table>
There are typically four to six problems to be solved by each team, so
the state diagram can be repeated for each problem to be solved.

The Hello message is sent by each COPS process to PCSINFO upon
first starting. PCSINFO verifies that the sender is authorized to
access the programming contest, and that this team is not already
logged in on another terminal. If the contestant's process meets
these criteria, he is granted access to the proctor system, and the
process will enter the READY(p) state. If the user is not authorized,
PCSINFO will ignore further communication. The signon process
prevents unauthorized multiple access by a team, but allows a COPS
process to be restarted in the event of serious error.

The PracticeRun message is issued by each team to the PCSINFO
process whenever a problem is run with practice data. The message
contains the team number of the team issuing the RUN command, and the
problem number being executed by the contestant. PCSINFO maintains a
count of practice runs for each problem and team, although the contest
administrator may choose not to penalize contestants for these runs.
The actual execution of the practice run is handled at the user
terminal; PCSINFO merely records the execution. On completion of the
practice run, the COPS process remains in the READY(p) state.

When a contestant requests that his program be run with contest
data, a ContestRun message is transmitted to PCSINFO. Like the
PracticeRun message class, ContestRun messages contain the number of
the team and problem to be tested.
Upon receiving the ContestRun message, PCSINFO must first lock the program being submitted against further editing. Accordingly, the COPS process is placed in LOCK(p) state for the submitted problem. Then, PCSINFO selects the appropriate batch script for the problem being executed, and submits the job to the GALAXY-10 batch processor for execution.

The JUDGE program uses the Proctor System Message Service to inform the contestant, through the COPS program, that his contest run has been completed. Two separate message types, called RunSuccess and RunFail are used, based on whether the contestant's program solved the problem correctly. Both message classes contain the problem number to which the message refers. Upon receiving a RunSuccess message, COPS
prints a completion message, such as: "Contest run for problem 2 was successful," and enters the FINISH(p) state. If all problems for a team have been completed successfully before the end of the contest (that is, if COPS is in FINISH(p) state for all problems), a congratulatory message is displayed, and the team is logged off the DECsystem-10.

The RunFail message is transmitted from the JUDGE program to a COPS process after a contest run has completed unsuccessfully. In addition to the problem number, the RunFail message contains an error code which can be one of the following:

1. COMPILATION - The program has syntactic errors resulting in compilation errors.

2. EXECUTION - The program did not completely or properly terminate execution.

3. OUTPUT - The program executed properly, but the output data were either incorrect, or improperly formatted.

When a RunFail message is generated, an appropriate error message is displayed on the contestant's terminal, such as: "Contest run for problem 2 was unsuccessful - output error," and COPS returns to READY(p) state.

After gaining experience with the PCS1 prototype system it was clear that a facility for broadcasting textual messages from the contest administrator to contestants was necessary. Such lengthy messages did not conform to the other message classes, which were only a few bytes in length. Furthermore, the transmission of lengthy
messages introduced concerns over excessive overhead. An alternate strategy for handling broadcast messages was devised.

A broadcast message is issued by using the BROADCAST command in the JUDGE program. The message is assigned a broadcast serial number, and is stored in the PCSINFO memory resident database. A message is then transmitted to each contestant's process, containing the broadcast serial number. The receiving process then retrieves the message text from the shared memory resident database, and displays it on the contestant's terminal. Since the receipt of a broadcast message does not alter a the state of a COPS process, the Broadcast message type is not shown in Figure 3.

Message System Implementation

The TOPS-10 operating system provides several ways for cooperating processes to exchange information. The two principal methods are the Inter-process Communications Facility (IPCF), and the Process Software Interrupt system (PSI).

IPCF is a flexible message handler which allows processes to exchange data packets of up to two kilobytes in length. A built-in queuing mechanism holds messages until the receiving process reads them. Further, user programs may trap to a special interrupt handling routines when an IPCF packet is received. The principal drawback to IPCF is that each process is limited in the number of outstanding messages it may send without having the messages read by the receiving process. If this "message-sending" quota is exceeded, the sender is
blocked from transmitting any more packets.

The Process Software Interrupt system is a facility provided by TOPS-10 whereby user programs may trap to specified subroutines based on some software event, such as an input/output completion or a program error. One such event that a user may trap is an inter-process "doorbell" interrupt, which one process may use to signal another. The doorbell interrupt allows for 18 bits of user data to be transmitted with the interrupt, along with the process number of the sending program. Unlike IPCF packets, inter-process doorbell interrupts are not flow-controlled, are not queued, and have no associated quotas. Because the doorbell interrupts are non-blocking operations, if the receiver has not enabled the doorbell condition, the transmitter may repeat the interrupt until the receiver is ready.

Due to potential difficulties associated with IPCF quotas, and the greater complexity of the IPCF system in general, the software interrupt doorbell facility was selected as the primary communications strategy, in the Programming Contest Proctor System. However, since the messaging protocol was designed independently of the communications medium, it was possible to incorporate alternate messaging software in the event of repeated failures of the inter-process doorbell mechanism.

**Message Retransmission Techniques**

Simulation testing was performed to determine an optimal
algorithm for retrying message transmissions in the event of a failure. It was found that messages generally fall into two categories. Most messages elicited little activity of the receiver, other than to increment a counter, modify one or more status flags, or type a message. These messages can usually be processed in approximately 40 milliseconds, including time for the operating system to restart the receiving task. The ContestRun message normally takes up to 500 milliseconds to be processed by PCSINFO, since PCSINFO must in turn exchange messages with the GALAXY-10 batch controller.

The retry scheme derived from this information, illustrated in Figure 4, was to first wait 50 milliseconds after a transmission failure before attempting to resend a message. This would allow enough time for processing of a single "short" message by the receiver. If the second transmission attempt also failed, another retry was performed after waiting 500 milliseconds. This procedure would be repeated up to ten times before being aborted.

In practice, this message retry scheme has worked well; partly due to the retry mechanism, but mostly by restricting the message service to only those functions necessary for the operation of the proctor system. A 47-team contest, for example, was observed to have 2178 messages transmitted during its four hour duration; an average of only eight messages per minute. It is not surprising that there were no messages aborted due to repeated retry failure.

The Proctor System is designed to minimize the chance of a message service error, but even an error in message transmission is
not fatal. If a message cannot be transmitted, an error message is
displayed on the user terminal, along with a request to retry the
failing command. If an automatic judging session is aborted due to a
message service failure, the run must be judged manually.

---

```
Begin
  retries := 10;
  SENDMESSAGE(messagetype,teamnumber,problemnumber,errorcode);
  While (error) and (retries > 0) do
    Begin
      Wait(50);
      SENDMESSAGE(messagetype,teamnumber,problemnumber,errorcode);
      If (error) then
        Begin
          Wait(500);
          SENDMESSAGE(messagetype,teamnumber,problemnumber,errorcode);
        End;
        retries := retries - 1;
      End;
      If (error) then ABORT("System Error - Message Send Failed!");
    End;
End;
```

Figure 4. Message Retry Algorithm
Due to the success of the present message handling mechanism, no message acknowledgment scheme has been developed for use in the Programming Contest Proctor System. There is an implicit acknowledgment of messages in that the operating system will not indicate a successful transmission of an inter-process doorbell unless the receiver has accepted the interrupt. Further, the simplicity of the message facility itself, and the limited domain of message generators ensure that only well-formed messages will be transmitted. A positive acknowledgment structure would also double the message traffic, and greatly complicate the message handling software itself.

Backup Messaging Strategies

A backup message handling mechanism was designed and written utilizing the Inter-process Communication Facility described earlier. The IPCF based message system makes use of the same data formats as the inter-process doorbell mechanism. The only difference between the two facilities is the TOPS-10 system service used to transmit and receive the messages. It was intended that should the principal doorbell mechanism fail even after several retries, that the IPCF facility would be called upon to ensure message delivery. Due to the success obtained with the inter-process doorbell facility, though, the backup system has never been utilized.
PCSINFO Memory Resident Database

In order to minimize the message traffic among different processes in the Programming Contest Proctor System, it was necessary to develop an alternate method for obtaining system-wide or process-specific information from the central PCSINFO memory-resident database. The solution, utilizing multiple segments of shared memory, exploits the flexibility of the TOPS-10 memory management architecture by allowing both program code and data to be shared among contest processes as needed.

TOPS-10 Memory Organization

Before exploring the details of the Proctor System software architecture, it is useful to describe the TOPS-10 memory segmentation scheme as viewed by the user process. TOPS-10 divides user address spaces logically into discrete segments, in a manner similar to, but less complex than that of the TOPS-201 or Multics2 operating systems.

Under TOPS-10, each process is allowed to have two contiguous areas of memory, called segments, in its address space. These segments are referred to as the low segment and high segment, due to their usual starting address of 000000g and 400000g respectively. The low segment is always privately owned by a single process, and cannot

---

1Bell, "The Evolution of the DECsystem-10", pp. 55-56.

be accessed by another process without special privileges combined with an intimate knowledge of DECsystem-10 virtual address mapping. The high segment, however, may be shared among many users and may be assigned several levels of access allowing reading, writing, or execution of data within the segment.

The most common use of shared high segments is having a program whose executable machine code resides in a shared segment, with the impure data for each process in the low segment. This enables several users to run the same program from the same physical memory, eliminating redundant memory resident copies of the program, and dramatically reducing the amount of process swapping performed by the operating system.

Another less common use for shared high segments is as a common data area. In this context, the executable code may reside either in the private low segment, or in shared high memory. The owning process, that is, the process which creates the high segment, may make the segment writable for itself, or for all processes. In this way, a common database may be created in memory for all processes to share. The database will remain intact even after all processes using the data have been stopped. The high segment can only be restored to its original state by superseding the program image on disk, or by reloading TOPS-10 altogether.

Proctor System Memory Organization

The Programming Contest Proctor System makes unusual use of
TOPS-10 memory management in that it uses shared high segments for both executable code and data storage. Figure 5 gives an overview of the memory structure of the Proctor System. The COPS command language and editor consists of a shared high segment, which contains most of the COPS program code. The size of this high segment is approximately fifteen kilowords. By sharing this memory, a contest with fifty competing teams can realize a memory savings of approximately 750 kilowords.

<table>
<thead>
<tr>
<th>COPS</th>
<th>Address</th>
<th>PCSINFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPS program code</td>
<td>777777</td>
<td>PCSINFO data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PCSINFO access routines</td>
</tr>
<tr>
<td></td>
<td>400xxx</td>
<td></td>
</tr>
<tr>
<td></td>
<td>400010</td>
<td></td>
</tr>
<tr>
<td>Op Sys segment data</td>
<td></td>
<td>Op Sys segment data</td>
</tr>
<tr>
<td>Start of COPS hiseg</td>
<td>400000</td>
<td>Start of PCSINFO hiseg</td>
</tr>
<tr>
<td>Top of lowseg</td>
<td>377777</td>
<td>Top of lowseg</td>
</tr>
<tr>
<td>COPS user data &amp;</td>
<td></td>
<td>PCSINFO program &amp;</td>
</tr>
<tr>
<td>Hiseg switching code</td>
<td></td>
<td>private data</td>
</tr>
<tr>
<td></td>
<td>000140</td>
<td></td>
</tr>
<tr>
<td>Job Data Area</td>
<td></td>
<td>Job Data Area</td>
</tr>
<tr>
<td></td>
<td>000017</td>
<td>Accumulators</td>
</tr>
<tr>
<td>Start of COPS lowseg</td>
<td>000000</td>
<td>Start of PCSINFO lowseg</td>
</tr>
</tbody>
</table>

Figure 5. Proctor System Memory Organization
On systems with a million words of storage or more, sharing of the COPS high segment can eliminate the swapping of contest related processes entirely.

The private COPS low segment contains user data, and a few routines to access data in the PCSINFO segment. Since TOPS-10 only allows a process to address one shared high segment at once, the COPS low segment routines must connect to the PCSINFO segment, collect the necessary data, and restore the COPS high segment. While there is some operating system overhead associated with this procedure, the cost is not great, since both high segments are almost guaranteed to be resident in memory.

The PCSINFO high segment is a part of the PCSINFO program itself. A major part of the PCSINFO initialization is to allow the PCSINFO high segment to be read by all Programming Contest System components, and to allow the segment to be modified by the JUDGE program or PCSINFO itself. The PCSFSS, PCSCOR, and PCSDPY programs are all executed in private low segments, and read data from the PCSINFO segment.

Data Shared Within the PCSINFO High Segment

Eight tables are defined within the Programming Contest Proctor System for common data storage. These tables are each referenced through an eight element vector of pointers at the beginning of the data area. All high segment data are accessed through a set of routines located within the PCSINFO high segment itself, and whose
addresses are bound at compilation time to 400010g-400014g. The PCSINFO data tables and access routines are shown in Table 4. Appendix D details the actual MACRO-10 statements used to generate the PCSINFO common database.

The system data table contains information of global significance to all contest components, such as the starting and finishing time of the contest, the serial number of the most recent broadcast message, and the job number of the PCSINFO master process.

Table 4
PCS Shared Database Description

<table>
<thead>
<tr>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SysData</td>
<td>System data table</td>
</tr>
<tr>
<td>ProbStatus</td>
<td>Status of each team's problems</td>
</tr>
<tr>
<td>PracRuns</td>
<td>Count of practice runs for each problem</td>
</tr>
<tr>
<td>ConRuns</td>
<td>Count of contest runs for each problem</td>
</tr>
<tr>
<td>FinishTime</td>
<td>Finish time of each problem</td>
</tr>
<tr>
<td>TeamPPN</td>
<td>Project-programmer number of each team</td>
</tr>
<tr>
<td>TeamJob</td>
<td>Job (process) number of each team</td>
</tr>
<tr>
<td>BcastList</td>
<td>Queue of recent broadcast messages</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The BcastList table is an array of eighty character strings containing the four most recent broadcast messages, which may be retrieved by COPS processes which may have been restarted or may have otherwise missed a broadcast message.

Both the TeamPPN and TeamJob tables are one-dimensional vectors containing the project-programmer number and job number of each team in the contest. These values are filled in by PCSINFO as a result of the COPS signon procedure, and are used to verify the authenticity of all messages received by PCSINFO.

The PracRuns, ConRuns, FinishTime, and ProbStatus tables are all two-dimensional arrays, indexed by team and problem number. PracRuns and ConRuns indicate the number of practice and contest runs for each problem, and FinishTime indicates the time of successful completion for the problem. The ProbStatus table indicates the completion status of each problem; either completed, not completed, or locked awaiting evaluation of a contest run. COPS uses this third state to indicate that the contestant's file is in the LOCK(P) state and cannot be edited until the contest run has been evaluated.

The shared database is highly structured, and is only large enough to accommodate the specific number of teams and problems desired by the contest manager. In order to achieve this level of space efficiency, the structure of the shared data area is bound at compilation time. If the manager wishes to change the number of teams allowed in the competition, or the number of problems to be solved, the entire Proctor System must be recompiled and linked.
CHAPTER V

TESTING AND USE OF THE PROCTOR SYSTEM

Simulation and User Testing

The development of the Programming Contest Proctor System was a gradual process, which allowed a considerable amount of use before the first large scale contest was held. Through standard testing methods and hands-on use by students in small local competitions, the COPS editor and command language interpreter was found to be a highly reliable program. It was also determined that if, for some reason, a contestant had to restart COPS, due to a damaged terminal or other failure, that COPS could be easily restarted. Further, the JUDGE program was exercised fully in local competitions, as were the ancillary Proctor System facilities. The distributed design of the proctor software confined load dependent performance concerns to the PCSINFO program itself. Of all the Programming Contest Proctor System modules, only PCSINFO could be affected by increasing the number of participants in the competition, and only a failure of PCSINFO could affect the entire Proctor System.

The performance of PCSINFO is directly related to the number of teams participating in the contest, since all COPS processes must communicate with PCSINFO when logging in or submitting a program for execution. In order to test the functions of PCSINFO under a heavy
load, it was necessary to create a series of test programs to send different types of messages to PCSINFO, both valid and invalid, in order to ensure that PCSINFO would accurately process well-formed messages, and reject those that were invalid.

This simulation procedure was later used in an actual local competition, where test programs were directed to send messages to PCSINFO at a high rate of speed while eight contestants participated in a normal programming contest. The test programs generated messages at a rate of approximately 20 messages per second, without interfering with the activities of the competitors. These simulations not only demonstrated the ability of PCSINFO to process messages correctly, but also aided in determining the speed of message handling by PCSINFO. This in turn led to a refined message retry mechanism discussed earlier.

Production Use of the Proctor System

On Saturday morning, November 18, 1984, Western Michigan University hosted the ACM East Central Regional Programming Contest. The Programming Contest Proctor System was used for staging this competition, which in which 47 teams participated representing colleges and universities from geographic locations as widespread as Kentucky, Pennsylvania, and the Canadian province of Ontario.

All teams registered in advance of the competition, and were sent user manuals describing the COPS editor and command language interface. In addition, descriptions of the FORTRAN and Pascal
languages on the DECsystem-10 were included. On the evening before the contest, the contestants had a brief opportunity to gain hands-on experience with the Proctor System software.

At the start of the contest, each team was given four problems to complete within four hours. At that time, the individual COPS processes were seeded to their respective terminals. Each team was assigned a single computer terminal for use during the competition. Seven independent batch streams were activated for processing contest runs.

Within minutes of the start of the contest, several contestants were at their terminals entering their solution programs. The graph in Figure 6 indicates that the DECsystem-10’s dual central processors ran at between 80 and 100 percent of capacity for the duration of the competition. Over 1600 practice runs were made by the teams, with 159 contest runs submitted. Over the four hour period, or some 200 user-hours, there were no errors detected in the operation of the Programming Contest Proctor System.

The graphs in Figures 6 and 7 meter three additional variables, besides central processor utilization: the number of jobs logged in, total amount of disk input and output performed, and the amount of disk input and output due to virtual memory activity and job swapping. Each figure is expressed as a percentage of the theoretical maximum.
Figure 6. Job and CPU Utilization During Nov. 18 Contest

Figure 7. Disk and Swapping Loads During Nov. 18 Contest
Some trends can be seen in the graphs. The CPU utilization shows two peaks; the first about midway through the competition, and the second near the end. Also at the midway point, there is a rise in the level of disk activity which increases throughout the rest of the contest. The number of jobs logged in remains constant within one or two jobs, until the last half hour of the contest where the count rises by over five jobs. From these graphs, the following conclusions can be drawn:

1. Contestant activity was largely confined to editing until about two hours into the contest, at which point most contestants had solution programs to run. At this point, there was a significant increase in executions and related CPU activity.

2. The second half of the contest was considerably busier than the first. The gradual increase in disk activity indicates a steady stream of both practice executions as well as contest runs.

3. The relatively stable job count indicates that until the last half hour of the competition, there were not many judged contest runs, and they were rather evenly spaced. At the end of the contest, though, most contestants tried last-minute contest runs, hoping to complete a successful run. As the contest shut down, all seven batch streams were active, and over 25 contest runs awaited execution.

4. For the duration of the contest, there was essentially no swapping of user processes from main memory to disk. This saved significant amounts of CPU time that would have otherwise have been spent in operating system overhead.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Analysis of Proctor System Performance

A major concern when designing the Programming Contest Proctor System was the efficient use of system resources; leaving sufficient processing reserves to allow the contestants to create and execute their programs without major delays. This goal contributed to the simplified messaging system, shared memory design, and modularity which provided users with a fully operational programming environment which was efficient and responsive.

Over the four hour duration of the November 18 contest, the PCSINFO process accounted for only 16 seconds of runtime. The JUDGE program used about 20 seconds of processor time, and each of three jobs running PCSDPY accumulated about a minute of runtime each. The failsafe process, PCSFSS, also recorded just over a minute of runtime.

The remaining central processor time, approximately 7.9 processor hours, were available for the contestants' processes, system processes such as the batch controller, and for operating system itself. The amount of processor time used by contestants averaged nearly two minutes per team. This accounts for the time spent both in the COPS editor and executing practice runs. The time spent within COPS was not recorded but can be approximated at 1.25 CPU minutes per process, based on execution counts and average execution times.

A runtime of 1.25 minutes per process is somewhat higher than desired. It was determined that much of the excess runtime accumulated by COPS was due to an "auto-save" feature incorporated into the COPS editor. After every fifth command, COPS automatically

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
updated the file on disk that was currently being edited, as a precaution against system failure. The practice of saving data after every five commands was undoubtedly overly cautious, and could be modified for greater efficiency.

Hardware Configuration Guidelines

Performance statistics gathered during and after the regional programming contest aided greatly in defining the hardware configuration required for successful operation of the Programming Contest Proctor System. In the regional competition, maximum utilization was made of the dual central processors, without significant delays being experienced by the contestants. With minor modifications to the COPS program outlined above, a similarly equipped DECsystem-10 could accommodate 50 to 55 competing teams comfortably. Based on runtime figures obtained during the contest, approximately 30 to 35 teams would fully saturate the resources of a single processor KL-10-based system.

Disk Requirements

The executable programs which comprise the Proctor System require a total disk space of approximately 700 blocks or roughly 450 kilobytes. During the programming contest itself, up to 5000 blocks of temporary space may be required for contestant programs, batch control files, and batch log files. If this much disk space is not available, the programming contest can run with as little as 1000
blocks of temporary file space by deleting batch log files as they are printed.

Terminal Usage

All of the 47 teams in the regional contest were connected to the DECsystem-10 through similar terminals at a data rate of 2400 bits per second. The terminals were connected to the DECsystem-10 via a Digital Equipment DN87S terminal concentrator, which is based upon a PDP-11/40 minicomputer. The DN87S is designed to accommodate 64 terminals at speeds up to 9600 bits per second. In most environments, it is assumed that no more than a few terminals will be used at any given time. The contestants made consistent use of 47 terminals, and proved to overload the single terminal controller, resulting in unacceptable delays in echoing of typed characters. Clearly, a second terminal concentrator would have alleviated this situation.

Problems Encountered by Contestants

The contestants adapted to the COPS environment remarkably well. In fact, there was only one problem that could be traced to a deficiency in the COPS user interface. On two occasions, contestants entered a solution program for one problem, after identifying the program as the solution for a different problem. The lack of a RENAME facility in COPS prevented the user from correcting his mistake and the files had to be renamed by the contest organizer. It was not an oversight to leave the RENAME facility out of COPS. Rather, it was
felt that the user should have no access to the underlying file system, and that a RENAME capability would add another command for the user to remember. A possible solution to this problem would be the addition of a "hidden" RENAME capability, which a contest organizer or aide can execute at the contestant's terminal.
CHAPTER VI

CONCLUSIONS

The success of any software project is best measured in the successful use of that software by a diverse group. In this respect, the Western Michigan University Programming Contest Proctor System has been a highly successful project. Moreover, although none of the backup failsafe software was exercised in an actual competition, this is seen as a further indication of the degree of success attained in this project. For though a software designer will always attempt to make software that can gracefully recover from any errors, it is more gratifying still when no errors occur from which the software must recover.

The Proctor System software has demonstrated the ability to have a tightly-linked, multi-process software system which is sharable and reentrant, under the control of the TOPS-10 operating system. Finally, the software developed for this project has demonstrated a reliable method for the automated administration of computer programming contest; providing the elements necessary to such a competition: a secure, easily-learned programming environment, an automated scoring facility, and a reliable software kernel.
Operational Results

The 1984 ACM East Central Regional Scholastic Programming Contest was one of the largest regional competitions ever to be held by the ACM. Despite this, the contest was carried out efficiently and successfully.

User Response

During the contest, each team was assigned a work area, and had an assigned terminal in one of several terminal rooms nearby. When a contestant submitted his contest run, he was instructed to return to his work area to await the results of his run. Frequently, the results would arrive in the work area ahead of the contestants, enabling the contestants to resume debugging activities immediately.

Display terminals running the PCSCOR program were placed near the contestant work areas, as well as in the advisors' reception area. These display terminals provided an immediate update of each team's progress in the competition. The display terminals were well received by both the participants and their advisors.

Lastly, the contest adhered to its schedule. It started and finished on time, and despite several dozen contest runs that were submitted at the close of the contest, all runs were evaluated, and the final scoresheet was printed only fifteen minutes after the competition's conclusion. This factor was especially appreciated by those teams who were faced with long journeys home.
User Participation

The successful operation of the ACM regional programming contest required a great deal of effort by many groups. Contest registration, facilities setup, and preparation of information packets demanded the resources of several dozen faculty, staff, and students of the university. The proctor software required the services of a contest administrator to configure and operate the software package, three judges, a judges' terminal operator, and assistants for each of four terminal clusters that served the competing teams.

Additional assistants were utilized to deliver contestants' output and execution results to their workrooms. The continuous use of the DECsystem-10 line printers by the contest necessitated the use of an additional input/output clerk. Clearly, the Programming Contest would not have been successful without this kind of support.

Future Research

The WMU Programming Contest Proctor System has successfully demonstrated the potential of automatic proctoring facilities in the programming contest environment. It can also serve as a starting point for future research into proctoring software.

Alternative Hardware Architectures

At the time the Programming Contest Proctor System was developed, the DECsystem-10 was the only computer available at Western Michigan University with the capacity to accommodate a large programming
As computing hardware has become less expensive and more powerful, several possibilities for alternative proctor system architectures have arisen.

**Networked Microcomputers**

One alternative to the mainframe proctor system architecture is a network of microcomputers, linked by a high-speed local area network. Under this design, each team would use a personal computer to create and test its solution programs. One microcomputer would be used as an "execution server," and would execute the solution programs with contest data. A "contest administrator" computer would be used to pass execution requests to the execution server, and to route results to a "judge" processor for evaluation. The organization of such a network is shown in Figure 8.

In a microcomputer-based contest, each element serves the same purpose as its counterpart in the mainframe contest architecture. The main difference here is that where each element was a user task under the timeshared operating system, each element of the microcomputer contest is an independent computer.

There are several advantages to such a structure. Chief among these is the amount of processing power afforded each user. Since each contestant team has a computer on its desktop, a more sophisticated and easier-to-use contestant interface can be developed.
Figure 8. Microcomputer-based Contest Organization

Another benefit to this structure is that it is less sensitive to increased loads imposed by additional contestants, since adding a contestant also adds a computer to the architecture.

The principal change in the proctor system design in a microcomputer-based contest would certainly be a more robust communications architecture. In the single machine TOPS-10 design, there was little chance of a single proctor system component not receiving a message. The possibility of lost messages is considerably increased in a networked environment.

In addition, the shared memory scheme used in the mainframe proctor system could not be used in the microcomputer environment. Instead, the shared information maintained by the contest administration module would have to be queried through extensions to the message facility.
Single or Networked Super-Minicomputers

Since the development of the Programming Contest Proctor System, several manufacturers have introduced medium-sized timesharing computers, often referred to as super-minicomputers. One example is the VAX\(^1,2\) family of computers from Digital Equipment Corporation. The VAX family includes over a dozen models which can support from one to more than a hundred users, depending on the tasks being performed. Two operating systems are available for the VAX: VAX/VMS and UNIX. Extensive networking software allows VAX systems to be linked into large computer networks.

While the largest VAX processors available at present could probably support a competition involving fifty teams, most VAX systems would not be able to support more than twenty to thirty teams. Figure 9 shows an alternative structure wherein a programming contest is conducted on a network of two computers. Under such a system, the responsibility of contest administration is shared between two "local" contest administrator modules. These modules would communicate with each other in order to maintain a common global contest database, giving the appearance of a single contest. In this way, contestant, judge, score-keeping and other processes may reside on any CPU participating in the contest.

---


2VAX and VAX/VMS are trademarks of Digital Equipment Corporation.
Figure 9. Network-based Contest Organization
It is interesting to note that in both the microcomputer-based contest and the super-minicomputer implementation, the overall structure of the contest remains the same as in the mainframe version, and very similar to the batch-mode contest model discussed in Chapter 2.

**Classroom Use of the Proctor System**

Although the Proctor System was well suited to the programming contest environment, it has not proved to be easily adaptable to the classroom setting for several reasons:

1. As it is currently structured, the Proctor System must be used simultaneously by all participants. It is difficult to reserve large numbers of computer terminals for this purpose in advance.

2. The Proctor System must be set up and started prior to use. The process of setting up contestant accounts on the DECsystem-10 and preparing the accounts for Proctor System access is time-consuming and laborious.

3. Many of the design considerations which were important in the contest setting do not apply in the classroom. For example, the user interface was designed to be easily learned by new users. In a classroom setting, it is assumed that the students will understand the host command interpreter and editor.

While there are many features of a proctoring system which are common to the classroom and the programming contest environments, there are sufficient differences in the goals to cast doubt on the applicability of a contest-oriented proctoring facility to classroom
use. Adapting the current proctor system designs to other uses and other computing environments would provide many avenues for future research.
Appendix A

COPS User Guide
COPS User Guide

Welcome to the WMU Programming Contest Proctor System. This system is designed to provide a simple, uniform programming environment for a large group of users. The Proctor System command processor, COPS, provides a simple editor and command language to remove any "home team" advantage to those familiar with the DECsystem-10 computer and its TOPS-10 operating system. The programming contest environment also provides automatic judging of contest runs, computation of scores, and display of team status.

This document describes the BASIC-style line-oriented editor in COPS, and the associated command language. Under this command language, as in most BASIC-style systems, a line of input is treated as a command unless it is preceded by a line number. Unlike most BASIC systems, commands may be abbreviated to the minimum length for which they are unique. Thus, "HELP" may be abbreviated "H", "LIST" may be shortened to "L", and so on.

After completion of any command other than a line insertion, you will be prompted with a message such as:

[Ready - T:1 P:2 Program] or [Ready - T:1 P:2 Data]

indicating your team number, problem number being edited, and whether you are editing your program or data.

<table>
<thead>
<tr>
<th>Command</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;line-num&gt;</td>
<td>Delete line &lt;line-num&gt;.</td>
</tr>
<tr>
<td>&lt;line-num&gt; &lt;text&gt;</td>
<td>Insert/replace text in &lt;line-num&gt;.</td>
</tr>
<tr>
<td>LIST</td>
<td>List lines in program or data.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete lines in program or data.</td>
</tr>
<tr>
<td>RESEQUENCE</td>
<td>Renumber program or data file.</td>
</tr>
<tr>
<td>PRINT</td>
<td>Print program or data on line printer.</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>Edit your program.</td>
</tr>
<tr>
<td>DATA</td>
<td>Edit your data.</td>
</tr>
<tr>
<td>PROBLEM</td>
<td>Change problem number you are editing.</td>
</tr>
<tr>
<td>NEW</td>
<td>Scrap current program and start over.</td>
</tr>
<tr>
<td>RUN</td>
<td>Run your program (practice or contest).</td>
</tr>
<tr>
<td>STATUS</td>
<td>Display status of your team.</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Line-Entry Control Characters

- `<rubout>` or `<delete>` Erase previous character.
- `<ctrl-U>` Erase current line.
- `<ctrl-W>` Erase previous word.
- `<ctrl-R>` Retype current line.
- `<return>` Send current line to system.
- `<ctrl-S>` Suspend typeout to your terminal.
- `<ctrl-Q>` Resume typeout to terminal.
- `<ctrl-C>` Abort execution of your program.
- `<ctrl-T>` Show execution status of your program.

Inserting, Deleting, and Replacing Lines of Text

A file is composed of one or more lines of text. Each line of text is identified with a LINE SEQUENCE NUMBER or LINE NUMBER. This applies to both programs and data. Line numbers are not the same thing as FORTRAN or Pascal statement numbers. To insert a line of text into your file, you must type the line number for the new line, followed by a space, followed by the text to be inserted.

Example:

```
10 This is a sample line of text.
```

Note that the space between the line number and the text does not become part of the text, thus in our "This is a sample..." line, the "T" is in column one of the file. Let us look at another example using a short FORTRAN program.

Example:

```
10 WRITE(5,1)
11 l FORMAT(IX,'FORTRAN is fun. ')
30 END
```

In the above example, the WRITE statement on line 10 refers to the FORMAT statement number 1. The FORMAT statement number 1 is on line 11. Notice that the statement number 1 is in column one, and the word FORMAT begins in column 7.

The DECsystem-10 FORTRAN-77 compiler allows for the use of tabs in source programs. The `<tab>` character will cause the compiler to "skip" the rest of the statement number field and start interpreting the rest of the statement. The COPS editor takes advantage of this feature by using the `<tab>` as a delimiter (as the spaces), but treating it as text as well.
Example:

```
10 WRITE(5,1)
11 FORMAT(1X,'FORTRAN is fun. ')
30 END
```

NOTE: When typing a line with a statement number, MAKE SURE you have a space separating the line number from the statement number. If you type "151 END" instead of "15 l END", you may accidentally delete a line.

To replace a line that has been entered, simply retype the line. The COPS editor will respond with the message: [Replaced].

Example:

```
11 FORMAT(1X,'FORTRAN is NOT fun. ')
[Replaced]
```

To delete a single line, type the line number followed by a carriage return. COPS will respond with: [Deleted].

Example:

```
11 [Deleted]
```

The DELETE command may also be used to delete one or more lines.
**DATA command**

**Purpose:** To edit data rather than the user program.

**Usage:** DATA

**Example:** DATA

[Ready - T:1 P:1 Data]

**DELETE command**

**Purpose:** To delete one or more lines of text.

**Usage:** DELETE lnum

or: DELETE lnum-lnum

**Where:**
- lnum is a single line to be deleted
- or: lnum-lnum is a range of lines (low to high) to be deleted

If more than 4 lines are to be deleted, COPS will respond with:

Is deletion of n lines OK? (Y or N)

where "n" is the number of lines to be deleted. If you indeed want to delete that many lines, type a "Y" followed by a return.

**Example:** DELETE 10

[1 line deleted]

Example: DELETE 10-40

Is deletion of 6 lines OK? (Y or N) Y

[6 lines deleted]
HELP command

Purpose: To type helpful information.

Usage: HELP
Or: HELP <command-name>

Typing HELP by itself will give a brief command summary.
Typing HELP followed by a command name will give a
description of the command.

Example: HELP LIST

LIST command

Purpose: To type one or more lines of a file.

Usage: LIST lnum
or: LIST lnum-lnum
or: LIST

Where:

<table>
<thead>
<tr>
<th>lnum</th>
<th>lnum-lnum</th>
<th>no parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>is a single line number</td>
<td>is a range (low-high)</td>
<td>indicates the whole file</td>
</tr>
</tbody>
</table>

Example: LIST 1-99999
Example: LIST 100

NOTE

Since line numbers can range from 1 to 99999, the above
command is equivalent to the LIST command with no parameter.
NEW command

Purpose: To start a NEW program.

Usage: NEW

COPS will respond with:

WARNING: This will DESTROY your current program!
ARE YOU SURE? (Y or N)

You must respond YES in order to proceed.

COPS will then ask:

FORTRAN or PASCAL?

Respond with the language you'll be using for this program, FORTRAN or PASCAL.

PRINT command

Purpose: To print a user file on the line printer

Usage: PRINT

A listing of your PROGRAM or DATA will be printed on the line printer, depending on whether you are in PROGRAM or DATA mode.

PROBLEM command

Purpose: To edit program or data for a different problem.

Usage: PROBLEM n

Where "n" is the number of problem to edit.

Example: PROBLEM 2

If you are just starting to do this problem, COPS will ask:

FORTRAN or PASCAL?

Respond with the name of the language you wish to use.
PROGRAM command

Purpose: To edit user program rather than data.

Usage: PROGRAM

Example: PROGRAM
          [Ready - Til Pti Program]

RESEQUENCE command

Purpose: To resequence a user's program or data line numbers.

Usage: RESEQUENCE

    The user file will be resequenced, beginning at line 00010,
    and continuing in increments of 10 through the whole file.
RUN command

Purpose: To compile and execute your program.

Usage: RUN

COPS will respond with:

Do you want a PRACTICE or CONTEST run? (P or C)

If you respond with a "P", your program will be executed on your terminal, and you will be charged with a practice run. If you specify a contest run by typing a "C", a batch job will be submitted to run your program with contest data. COPS will not permit you to edit your files until the contest run has been completed. It will prompt for a new problem, and another team member will be allowed to use the terminal to work on another problem.

When your contest run has completed, a message will appear on your terminal indicating either that your program has executed successfully, or that the run was unsuccessful for one of the following reasons:

- COMPILATION - Program had compilation errors.
- RUNTIME - Program terminated with an execution error.
- OUTPUT - Program had an output error - the answers were wrong, missing, or badly formatted.

Example: RUN

Do you want a PRACTICE or CONTEST run? (P or C) P
FORTRAN: MAIN
LINK: Loading
[LNXCT 000000 Execution]

(Program output would be typed here)

End of Execution
CPU TIME: 0:0:1 ELAPSED: 0:0:5
[Ready T1l P1l Program]
Example: RUN
Do you want a PRACTICE or CONTEST run? (P or C) C
[Contest run submitted]
Which problem? 2
[Ready T:1 P:2 Program]

**STATUS command**

Purpose: To display status of the programming team.

Usage: STATUS

For each problem to be solved in the contest, the display will show the problem number, status, and number of contest and practice runs. The meaning of status conditions are as follows:

- **Done** Successfully completed program.
- **Active** This program being edited now.
- **Locked** Waiting for contest run to complete.
- **Idle** Program not active, but not completed.

Example: STATUS

<table>
<thead>
<tr>
<th>Problem</th>
<th>Status</th>
<th>Contest</th>
<th>Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Active</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Locked</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Idle</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Idle</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

[Ready - T:1 P:1 Program]
Appendix B

Proctor System Symbol Definition File
% PCSSYM - Symbol definitions for Programming contest system

This incredibly clever code is compiled by both the MACRO-10 assembler and the SAIL compiler in the form of "include files" using a common DEF (define) macro. %;

% Define the DEF macro - first for MACRO.
(This section is ignored by SAIL.) %

Universal PCSSYM - Symbol definitions for WMU PCPS.
SALL
Directive .nobin, flblst

Define DEF(sym,value,dscr),<sym=D'value>

% Define the DEF macro for SAIL.
(This definition is ignored by MACRO-10.);

Define DEF(sym,value,dscr)=<Define sym=value> ;

%******** End of definitions ******** %;

%*** The remainder of this code is compiled by both ***
*** MACRO-10 and SAIL. *** %;

%****** Site dependent constants ****** %;

Def(TEAMAX,30,<Maximum number of teams>);
Def(USRMAX,1, <Programmers per team>);
Def(TIMLIM,10,<Time limit per run>);
Def(FTIPCF,0,<We don't currently use IPCF>);

% *** Message type codes *** %;

Def(MS.BGN,0,<Contest begins in N minutes>);
Def(MS.FIN,1,<Contest finishes in N minutes>);
Def(MS.UNL,2,<Unlock unsuccessful run>);
Def(MS.DON,3,<Program done successfully>);
Def(MS.HEL,4,<Hello. I just started>);
Def(MS.SPR,5,<Submit practice run>);
Def(MS.SCR,6,<Submit contest run>);
Def(MS.BDC,7,<Broadcast message from judge>);
Def(MS.END,7,<End of contest (INF to self)>);
Def(MS.MAX,7,<Highest MS.??? code>);
Def(MS.POS,12,<Position in halfword of message code>);
Def(MS.WHO,1023,<Mask for user ID part of message>);
% *** User status codes *** %
Def(ST.IDL, 0,<Idle>);
Def(ST.ACT, 1,<Active>);
Def(ST.LOK, 2,<Locked>);
Def(ST.DON, 3,<Done>);

% *** Table names for hiseg data tables *** %
Def(T.COMM, 0,<Common data for system>);
Def(T.STAT, 1,<Status of each team>);
Def(T.PRUNS, 2,<Number of practice runs for each contestant>);
Def(T.CRUNS, 3,<Number of contest runs for each contestant>);
Def(T.FINTIM, 4,<Finish time of each contestant>);
Def(T.PPN, 5,<PPN of each team>);
Def(T.JOB, 6,<Job number of each team>);
Def(T.BCST, 7,<Pointers to broadcast messages>);

% *** Offsets into common data area *** %
Def(COM.MJ, 0,<Master job number>);
Def(COM.ST, 1,<Start time>);
Def(COM.FT, 2,<Finish time>);
Def(COM.BN, 3,<Current broadcast number>);
Def(COM.PD, 4,<PCS device>);
Def(COM.PP, 5,<PCS system PPN>);
Def(COM.PS, 6,<First SFD of PCS path>);
Def(COM.IC, 13,<Count of interrupts handled so far>);
Def(COM.IT, 14,<Time spent processing interrupts>);

% Now, provide an END statement for MACRO-10. %
END

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Appendix C

Proctor System Contest Administrator's Reference Guide

87
Objective - This guide is intended to aid the administrator in the operation of the WMU Programming Contest Proctor System.

References - This document assumes a good working knowledge of the DECsystem-10 and its system utilities. For further information about these items, please consult the following DECsystem-10 reference manuals: Operating System Commands Manual, Utilities Manual, and MIC Macro Interpreted Commands.

Obtaining Accounts - Obtain accounts from the DECsystem-10 system manager. One account is needed per programming team. One additional account should be obtained for the Proctor System itself. This account will henceforth be referred to as PCPSMASTER. The PCPSMASTER account should have a disk allocation of at least 5000 blocks. All the contestant accounts should be in the same project as PCPSMASTER. In this document, we will refer to this project as PCPSPROJECT.

Installing RUNCLI - The system manager must install RUNCLI.EXE on the SYS: area, and place RUNCLI in the list of JACCT privileged programs within the TOPS-10 monitor. The file SYS:RUNCLI.SYS must contain a line of the form:

```
COPS.EXE[PCPSMASTER]=[PCPSPROJECT,*]
```

Building the Proctor System - The program PCSGEN is used to rebuild the executable binary programs for the Proctor System. When you run PCSGEN, it will ask for the maximum number of competing teams, and the number of contestants per team. PCSGEN will generate a file called PCSCNF containing these values. Next, PCSGEN will ask if you wish to rebuild the Proctor System. If you answer "YES", the proctor system programs will be recompiled and linked.

Defining the TEAMS.PCS File - The file TEAMS.PCS is used for identifying each team in the Proctor System. To generate TEAMS.PCS, run the program PCSSET. PCSSET will first ask for the number of teams competing. Then, for each team, PCSSET will display the team number and account number, then ask for the team name. Any name, up to 20 characters may be entered. PCSSET will generate the TEAMS.PCS file in the proper format.
Preparing Contestant Accounts for Use - Two files must reside in each contestant's disk area before the contest is started. The first file, SWITCH.INI, gives initialization profiles for the system LOGIN program. SWITCH.INI must have a line of the following form:

```
LOGIN/DEFPRO:477/RUN:SYS:RUNCLI
```

The file ACCESS.USR is used to specify file access privileges for the user. Each contestant's ACCESS.USR file must have the following entry:

```
*.*/ALL=[PCPSMASTER],[*,*]/NONE
```

Preparing Contest Data Files - In order for the automatic judging features of the Proctor System to be used, the contest administrator must place the input data, and the correct output data for each contest problem in files on the [PCPSMASTER] account. The input data files are named FORnn.DAT, where "nn" is the problem number plus 20 (e.g., the input data for problem 1 is FOR21.DAT). The output data files are named ANSWER.xx, where "xx" is the problem number.

Starting the Proctor System - Before starting a programming contest, it is necessary to start the central Proctor System components: PCSINFO, PCSFSS, and JUDGE. While PCSINFO and PCSFSS are not interactive programs, they do occasionally display information on the controlling terminal. Thus, one must use the utility program OPSER to control all three programs. Following is an OPSER script to start the Proctor System. It first starts PCSINFO, then PCSFSS, then JUDGE. It also tells the JUDGE program to generate the batch control files necessary to execute contest runs for each contestant. Finally it sets start and finish times for the contest.

```bash
:login
:define info=
info-run pcsinfo
:login
:define fss=
fss-run pcsfss
:login
:define judge=
judge-run judge
judge-ctl
judge-start
judge-9:00
   | for 9:00 AM.
judge-finish
   | Set finish time
judge-13:00
   | for 1:00 PM.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Restarting After A System Crash - If the DECSystem-10 crashes during a contest, it will be necessary to restart the Proctor System. To do so requires only that you execute the same procedure as above. When PCSINFO restarts, it will indicate that PCSINFO is rebuilding the system database from a system crash. This information is located in the file PCSFSS.DAT, and must be deleted at the conclusion of the contest.

Judging Contest Runs - When a contestant submits a contest run for execution, it will be processed by the GALAXY-10 batch system. If the solution program fails to compile, fails to execute, or runs with the correct answer, the contestant will be notified automatically. If the solution program runs successfully, but the answer does not match the standard output key file, the contest administrator must judge the program manually. After evaluating the batch log file for the particular program, the run may be judged as successful or unsuccessful in the following manner:

Command: judge
Team: 1
Problem: 1
Was Run Successful(Y or N)? n
Error Type (Compile, Execute, Output)? o

Finishing the Contest - When a programming contest's time has expired, all contestant jobs will be logged off the DECSystem-10 automatically. Before logging out the PCSINFO job, you should print a final scoresheet by running the program PCSCOR. The final tabulation will be printed on the file SCORE.PCS. After verifying the scoresheet, all the Proctor System control jobs can be logged out by issuing the OPSER command:

:KILL ALL

JUDGE Program Command Summary - Following is a summary of the commands available in the JUDGE program. Each command is entered as a single command word. The operator is prompted for all further input, in order to minimize the possibility of entry error.
CTL command

Purpose: To generate batch control files for evaluating contest runs.

Additional Prompts: None.

Example:

Command: CTL
[Please wait... Generating control files]

START command

Purpose: To set the start time for the contest.

Additional Prompts: "Start Time?" prompt requests starting time in 24 hour format.

Example:

Command: START
Start time: 9:00
Start time is 9:00 AM (09:00).

FINISH command

Purpose: To set the finish time for the contest.

Additional Prompts: "Finish Time?" prompt requests starting time in 24 hour format.

Example:

Command: FINISH
Finish time: 13:00
Finish time is 1:00 PM (13:00).
JUDGE command

Purpose: To manually judge a contest run.

Additional Prompts: "Team?" prompt requests team number. "Problem?" prompt requests problem number. "Was run successful (Y or N)?" prompt asks if run was successful. A Y (yes) or N (no) response is expected. "Error type (Compile, Execution, Output)?" is asked if a run was not successful. A response of C, E, or O indicates the type of error.

Example:

Command: JUDGE
Team? 1
Problem? 2
Was run successful (Y or N)? N
Error type (Compile, Execution, Output)? O

BROADCAST command

Purpose: To broadcast a message to all contestant teams.

Additional Prompts: "Message:" prompt requests a single line message to be broadcast.

Example:

Command: BROADCAST
Message: The contest finish time has been extended to 1:00 PM.
[Message has been broadcast to all users.]
Appendix D

PCSINFO Shared Data Segment With Access Routines
TITLE PCSINFO - Info segment for programming contest system.

SEARCH UUOSYM,PSIMAC,PCSSYM ; Our own little symbols.
SALL
.Directional FLBLST

CHAN==>1
SALL
TW0SE6
RELOC 400000
LOC 400010

; Entry points for SPY, POKE, etc. SPYs use PORTAL to allow
; reading by all components, while POKES can only be done by
; PCSINFO itself.

ENTRY SPY
SPY: PORTAL SPY.l ; Entry for SPY.
POKE: JRST POKE.l ; Entry for POKE.
TYPEBR: PORTAL TYPE.B ; Entry for TypeBroadcast.
POKEBR: JRST POKE.B ; Entry for POKEBRoadcast.

;Internal Integer procedure SPY(Integer table, offset);
;Really a GETTAB-like function- return contents of table and offset.

SPY.l: MOVE 1,-2(17) ; Get table name
MOVE 1,SEGTAB(1) ; Get location of table
ADD 1,-1(17) ; Add offset to our entry
MOVE 1,(1) ; Get the value.
SUB 17,[3,,3] ; Fixup stack.
JRST @3(17) ; And do funky return.
; Internal procedure POKE(Integer table, index, value);
; Place a value in a shared data table

POKE.1: MOVE 1,-3(17) ; Table number.
MOVE 1,SEGTAB(1) ; Table address.
ADD 1,-2(17) ; Add offset into table.
MOVE 2,-1(17) ; Value to poke.
MOVEM 2,(1) ; Do it. Ill mem ref
; if segment locked!
SUB 17,[4,,4] ; Clean up stack.
JRST @4(17) ; Return.

; Internal procedure POKEBR(Integer table, index, value);
; Place a value in a broadcast table.

POKE.B: MOVE 1,-3(17) ; Table number.
MOVE 1,BCASTS(1) ; Table address.
ADD 1,-2(17) ; Add offset into table.
MOVE 2,-1(17) ; Value to poke.
MOVEM 2,(1) ; Do it. Ill mem ref
SUB 17,[4,,4] ; Clean up stack.
JRST @4(17) ; Return.

; Integer procedure TypeBroadcast(Integer BCnumber);
; Type the contents of judge's broadcast message;

TYPE.B: PUSH P,T1 ; Save a T...
MOVE T1,-2(P) ; Get broadcast number.
ANDI T1,3 ; Scale 0-3.
OUTSTR @BCASTS(T1) ; Type the broadcast msg.
MSG <>
POP P,T1 ; Restore T1,
SUB P,[2,,2] ; Cleanup stack.
JRST @2(P) ; And return Sail's way.
; ***** COMMON DATA AREA *****

; This table points to the PCS common data tables. All tables, 
; except the first, (COMDAT), are indexed on team number.

SEGTAB: EXP COMDAT       ; Common data.
        EXP PRGSTS       ; Programmer status.
        EXP PRUNS       ; Number of practice runs.
        EXP CRUNS       ; Number of contest runs.
        EXP FINTIM      ; Finish time for each user.
        EXP PPNTAB      ; TEAM/PPN translation
        EXP TEAM        ; TEAM/JOB translation
        EXP BCASTS      ; Pointers to broadcast msgs.

COMDAT: 0                     ; Master job no, initially 0.
     -1 and start time
     -1 and finish time
     0 Broadcast number
     BLOCK 7 PCSPPN, PCSDEV, and 5 SFDs.
     BLOCK 2 Interrupt accounting data.
     BLOCK 5 A little extra room.

BCASTS: EXP BC0               ; Table of pointers to
        EXP BC1               ; broadcasts.
        EXP BC2               ;
        EXP BC3               ;

BC0: BLOCK D80               ; Broadcast messages.
BC1: BLOCK D80
BC2: BLOCK D80
BC3: BLOCK D80

TMBLK==TEAMAX+1

PRGSTS: BLOCK TMBLK*USRMAX    ; Program status word.
PRUNS:  BLOCK TMBLK*USRMAX    ; Number of Practice runs.
CRUNS:  BLOCK TMBLK*USRMAX    ; Number of Contest runs.
FINTIM: BLOCK TMBLK*USRMAX    ; Finish (elapsed) time for
                              ; successful run.
PPNTAB: BLOCK TMBLK          ; TEAM/PPN translation
TEAM: BLOCK TMBLK            ; TEAM/JOB translation

END

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
BIBLIOGRAPHY


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


