BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

Report No. 2190
Job No. 11546

AFOSR - TR - 71 - 2845

INFORMATION PROCESSING MODELS AND COMPUTER AIDS FOR HUMAN PERFORMANCE

SEMIANNUAL TECHNICAL REPORT NO. 1, SECTION 2

TASK 2: HUMAN-COMPUTER INTERACTION MODELS

30 JUNE 1971

ARPA ORDER NO. 890, Amendment No. 6

Sponsored by the Advanced Research Projects Agency, Department of Defense, under Air Force Office of Scientific Research Contract F44620-71-C-0065

Prepared for:

Air Force Office of Scientific Research
1400 Wilson Boulevard
Arlington, Virginia 22209

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author)

Bolt Beranek and Newman
50 Moulton Street
Cambridge, Massachusetts 02138

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

3 REPORT TITLE

INFORMATION PROCESSING MODELS AND COMPUTER AIDS FOR HUMAN PERFORMANCE TASK 2: HUMAN-COMPUTER INTERACTION MODELS

4 DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Interim

5 AUTHOR(S) (First name, middle initial, last name)

Mario C. Grignetti, Duncan C. Miller, Raymond S. Nickerson, Richard W. Pew

6 REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
30 June 1971	95	8

8a. CONTRACT OR GRANT NO	9a. ORIGINATOR'S REPORT NUMBER(S)
F44620-71-C-0065	
b. PROJECT NO 890	
c. 61101D	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d. 681313	AFOSR - TR - 71-2845

10 DISTRIBUTION STATEMENT

Approved for public release;
distribution unlimited.

11 SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
TECH, OTHER	AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (NL) 1400 WILSON BLVD ARLINGTON, VIRGINIA 22209

13 ABSTRACT

We have implemented a Measuring System to obtain the statistical parameters necessary to specify a Queueing Theory model of the dynamic behavior of a state-of-the-art time-shared computer system, and present results on the statistics of usage of one such computer system.

We present a methodology for the performance of experiments involving human users and for the interpretation of their results. We expect that these results will yield predictive models for the overall efficiency of the "users-computer system" under various circumstances.

A paper has been prepared for publication describing the features that a system should incorporate in order to be considered effective and well human engineered.

DD FORM 1 NOV 68 1473

UNCLASSIFIED

Security Classification

INFORMATION PROCESSING MODELS AND COMPUTER AIDS FOR HUMAN PERFORMANCE

SEMIANNUAL TECHNICAL REPORT NO. 1, SECTION 2

Task 2: HUMAN-COMPUTER INTERACTION MODELS

30 June 1971

by

Mario C. Grignetti
Duncan C. Miller
Raymond S. Nickerson
Richard W. Pew

ARPA Order No. 890, Amendment No. 6
Sponsored by the Advanced Research Projects Agency, Department of Defence, under Air Force Office of Scientific Research Contract No. F44620-71-C-0065

Prepared for

Air Force Office of Scientific Research
1400 Wilson Boulevard
Arlington, Virginia 22209

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TASK 2: HUMAN-COMPUTER INTERACTION MODELS

1. Technical Problem

The purpose of this research program is to continue the development of models for human-computer interaction at the human-computer interface level.

2. General Methodology

Laboratory experiments.

3. Technical Results

We have implemented a Measuring System to obtain the statistical parameters necessary to specify a Queueing Theory model of the dynamic behavior of a state-of-the-art time-shared computer system, and we present results on the statistics of usage of one such computer system.

We present a methodology for the performance of experiments involving human users and for the interpretation of their results. We expect that these results will yield predictive models for the overall efficiency of the "users-computer system" under various circumstances.

A paper has been prepared for publication describing the features that a system should incorporate in order to be considered effective and well human engineered.

4. Department of Defense Implications

Large savings in the cost of software development are potentially possible by converting from the batch-processing computer

SEMIANNUAL TECHNICAL REPORT NO. 1, SECTION 2

Period 1 January 1971 - 30 June 1971

ARPA order No. 890
Program Code No. 1D20
Contractor: Bolt Beranek and Newman Inc.
Effective Date of Contract: 1 January 1971
Contract Expiration Date: 31 December 1971
Amount of Contract: $340,461
Principal Investigators: John A. Swets
Daniel N. Kalikow
Mario C. Grignetti
Duncan C. Miller
Telephone No. (617)491-1850
Title: INFORMATION PROCESSING MODELS AND COMPUTER AIDS FOR HUMAN PERFORMANCE

systems that are widely used today, tc interactive, time-shared computer systems. To design, operate, or even select an interactive system in a rational way, it is necessary to predict its relative acceptability and performance.

PREFACE

The present contract is a partial continuation of a research program begun in 1966 under ARPA sponsorship. Of the four tasks eventually funded under Contract F44620-67-C-0033, with the Air Force Office of Scientific Research, the first two tasks were awarded continuing support under the present contract. Those tasks are:

1. Second-language learning
2. Models of man-computer interaction

The present Semiannual Technical report covers the progress made in the second of these tasks during the first six months of the new contract. We have bound the reports of the two tasks separately to facilitate their distribution and use. In addition to a copy of this page, both sections of this report contain an appropriate subset of the documentation data required for the whole report: a contract information page, a summary sheet for the particular task at hand, and a DD form 1473 for document control

1. INTRODUCTION

In this progress report, we present the results of work on User-Computer Interaction performed from 1 February 1971 to 30 June 1971. The body of the report is organized as follows:

In Section 2, we deal with the subject of modelling the dynamic behavior of programs in a time-shared computer system. We give a succinct description of TENEX (the time-sharing operating system that we are using); we present a Queueing Theory model; and we describe the measuring system we have implemented to obtain the necessary statistical parameters.

In Section 3, we present several results on the statistics of session duration and actual computer time used, as well as on certain characteristics of system and subsystem performance.

Section 4 describes our work in the area we consider most difficult——that of modelling user behavior at the user-computer interface. As a result of this work, we believe that we have found a sound methodological basis for the performance of experiments and for the interpretation of their results, that will yield predictive models for the overall efficiency of the "users-computer system" under various circumstances.

Finally, in Section 5, we describe those system features developed at BBN and elsewhere that have turned out to be well human-engineered and particularly effective as user aids.

2. COMPUTER MODELS

2.1 INTRODUCTION

In this section, we shall describe our work towards the construction of probabilistic models for the dynamic behavior of programs in a time-shared computer. The models we seek are normative—we must be able to predict the effect of changes in the computer's configuration or of the user's demands on the system's response characteristics (see Section 4 for an overall view of our user-computer interaction modelling philosophy).

Probabilistic models based on Queueing Theory have been used with success in the past to describe the dynamic behavior of programs in a time-sharing system. The mathematical framework of Queueing Theory, with its treatment of units and servers, is a natural and legitimate body of knowledge upon which to draw for the construction of models. In fact, in a time-sharing system, user programs line up to be run one at a time (serviced by the central processor unit) until a termination condition is reached, whereupon they may undergo service by some other processor (server) and eventually return to the first server, all in rapid succession.

In the body of this section, we shall demonstrate the formal adequacy of such an approach for the TENEX system, and shall describe the measuring system that must be implemented in order to gather the statistics that will yield the model parameters.

2.2 MODEL STRUCTURE

The computer system we shall model is TENEX, a time-sharing operating system conceived and developed by BBN,* now available on two independent DEC PDP-10 computers at our Research Computer Center. The advanced features of TENEX, the availability of the systems personnel responsible for its development, the possibility of introducing changes in the operating system to meet measuring requirements, and the richness and variety of the user's environment at BBN are just a few of the reasons that make TENEX an obvious choice for our modelling efforts.

2.2.1 The TENEX System

TENEX is a system which utilizes paged core memory. In contrast to the swapping-type monitors like DEC's 10/40 or 10/50 monitors, TENEX allows users to write their programs as if they had a large (virtual) memory at their disposal, while at the same time reducing the time it takes to swap a user's program between core memory and secondary storage. This is so because only the working pages of a user's program (the "working set") need to be in core for his program to run. The necessary paging hardware—designed and built by BBN—makes it possible for core memory to be used more efficiently. Pieces (i.e., pages) of programs may be scattered anywhere in real core; the pager relocates each page to provide a contiguous "virtual memory" for the user. Thus, the system no longer has to worry about collecing "holes" in core memory (as is required in most non-paged systems) in order to fit programs in a simply-connected area of

*With the joint support of BBN and of the Advanced Research Projects Agency of the DOD.

real core. Another advantage of paging is that it makes it possible to run programs which would physically require more core than is available. In fact, only pages that are needed at the moment must be in core. When new pages that are not in real core are referenced they can be swapped in from secondary storage and the program can then continue execution. Note also that being able to run partially loaded programs can substantially increase core memory utilization.

Communication with TENEX takes the form of a dialogue in which the user gives a command, TENEX performs the desired action, and then waits for a new command. The collection of available commands, together with certain special characters and conventions, makes up what is known as the Executive Language, which is the user's handle on the time-sharing system. The language is very powerful and yet very easy to use, thanks to its good human engineering design. It is based on highly natural mnemonic commands and allows command recognition, input editing, and multiple input formats to be freely intermingled.

TENEX has a flexible file system. Files are distinguished by device, directory name, file name, extension, and version. Names and extensions may be up to 39 characters long. A very well human-engineered set of default values makes it extremely easy to reference commonly used files. Users can have several directories, and an elaborate system for file sharing and protection has been developed.

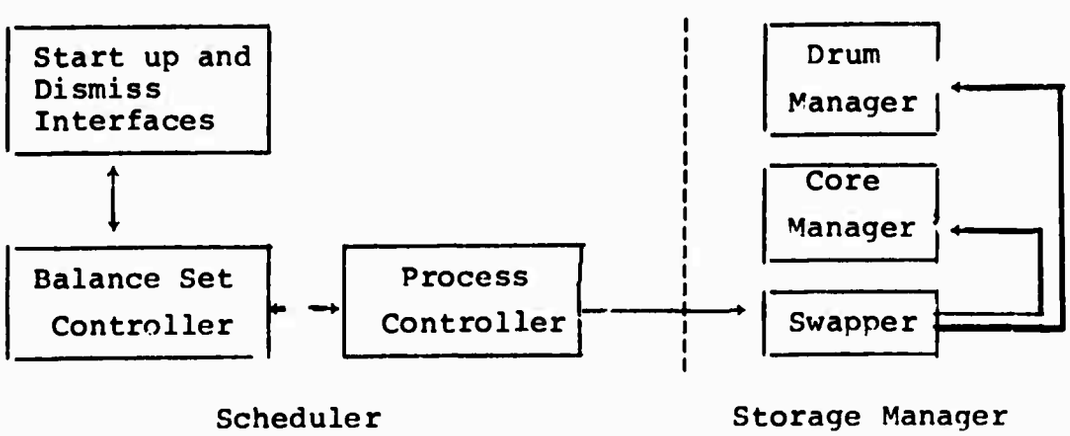
TENEX allows its users to run hierarchically dependent "parallel processes" that share memory among themselves and use a pseudo-interrupt system to facilitate interprocess communication.

Most standard user programs that run under the standard DEC PDP-10 operating system will also run under TENEX. Among them we have FORTRAN IV, MACRO and FAIL (machine language assemblers), LOADER, TECO (a powerful editing language), DDT (DEC's debugging language), TELCOMP (a BBN-developed language patterned after JOSS), LISP, and a variety of other subsystems of less widespread use.

2.2.2 Scheduling and Storage Management

A description of the structure of the TENEX software would be quite voluminous and is clearly beyond the scope of the present report. However, in order to be able to interpret and understand the structure of our model, it is necessary to describe at least the Scheduling and Core Managing functions. The following paragraphs are taken from TENEX memo #12.

> "The functions of Scheduling and Storage Managing are handled by several inter-related software modules, each with a specific, separable set of operations to perform.



> The modules to the left of the dashed line comprise the scheduler, those to the right the storage manager."

"The process controller performs those functions usually associated with a time sharing scheduler. It contains tables of all processes existing in the system and their state of execution (runnable, blocked for I/O, etc/). It contains routines which change the state of processes on request from other system modules or as a result of process activity. A central routine of the process controller performs the basic scheduling function, i.e., it considers the state of the processes in existence and the available system resources, and selects a process to be given some CPU service. It keeps an accounting of the recent activity of each process, particularly CPU usage, and allocates each system resource among the process competing for it according to some defined criteria."

"The balance set control is concerned with making efficient use of the core and drum channel resources of the system. It constantly monitors the state of core utilization and working set requirements of the processes in core, and decides when another process can be admitted or one must be thrown out. The "balance set" is defined as a set of runnable processes whose working sets can co-exist in core. It is thus a subset of the set of all runnable processes, and normally consists of those runnable processes which are most due for CPU service as determined by the process controller."

"The information gathering and decision making procedures involved in determining working sets and core utilization

are quite complex, and incorrect handling of these functions in a multi-process paged system can result in poor efficiency and bad service. The first step in avoiding this pitfall is to define a portion of the monitor which is directly responsible for these functions rather than having them diffused through many parts of the system."

"The function of the startup and dismiss routines is fairly common and straight forward. Included in this section are routines to save and restore environments as they go out of and into execution. No important scheduling or other decisions are made by this module."

"The swapper handles the communication between the secondary storage devices (drum and disk) and core memory. It receives requests from the scheduler to move processes into and out of core, constructs I/O requests and performs queueing.

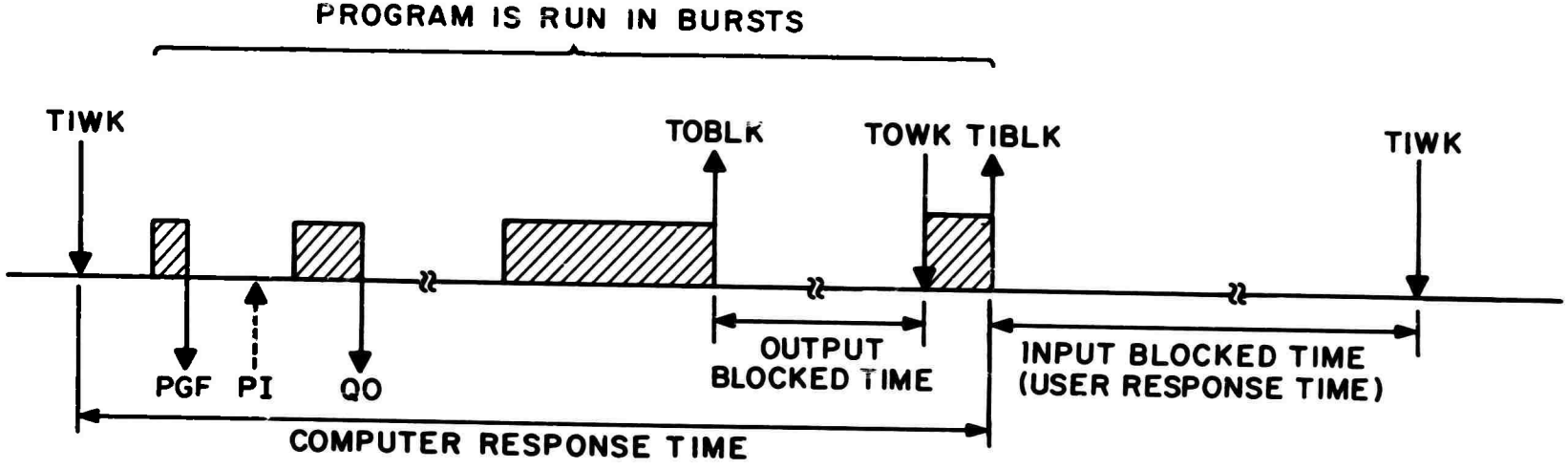
The core manager selects core pages to be used for swap reads from the drum or disk, performs some "aging" operations, and handles the selection of core pages to be swapped to the drum. It has principal use and control of the Core Status Table (CST) which reflects at all times the current state of each page of core memory. The CST is also modified by the paging hardware, recording information about the activity of the running process.

The drum manager is responsible for assigning storage on the swapping drum and for selecting pages to be moved to the disk in the event the drum becomes full."

2.2.3 The Basic Sequence of Events

Let us now consider a typical sequence of events as they would appear to a user when he gives a command to TENEX. Consider Fig. 1. The user types in the last character of his command (TIWK) which is usually a carriage return meaning, "now go and do what I command." The system recognizes such wake-up characters, and as soon as one is received the user's program becomes runnable. After some time, that depends on the system's load and the user's priority, the program becomes a member of the "balance set" and the CPU starts executing the given command until the user's program references a page that is not in real-core memory at the time; i.e., a page fault occurs (PGF). A request to read the page from the drum is entered after the core manager has found room for the page. Eventually the page is brought in (PI) and execution resumes. After possibly many such faults, the running time exceeds a fixed "quantum" (QO) and the program is dismissed (it is removed from the balance set). After some time (again, depending upon system load and upon a now dimished priority) execution continues and an output to be typed out on the user's terminal is generated. Execution stops and the program is dismissed as soon as the output buffer fills up (TOBLK). When the output buffer is almost empty, the program is reactivated (TOWK), generates the rest of the output (without filling up the remainder of the output buffer) and seeks further input from the terminal. Since the user has not yet typed in a wake-up character (he may not have started typing in his next command) the program is dismissed (TIBLK).

Let us next write a scenario for the sequence of events that occurs in scheduling and managing core for several processes. In



TIWK – TERMINAL INPUT WAKE-UP. PROGRAM BECOMES RUNNABLE AGAIN.
PGF – PAGE FAULT. PROGRAM REFERENCES PAGE THAT IS NOT IN CORE
PI – PAGE REQUESTED HAS BEEN SWAPPED INTO CORE
QO – QUANTUM OVERFLOW. PROGRAM IS DISMISSED
TOBLK – TERMINAL OUTPUT BLOCK - PROGRAM IS DISMISSED UNTIL OUTPUT BUFFER IS (NEARLY) EMPTY
TOWK – TERMINAL OUTPUT WAKE-UP. OUTPUT BUFFER IS (ALMOST) EMPTY AND PROGRAM BECOMES RUNNABLE AGAIN
TiBLK – TERMINAL INPUT BLOCK - PROGRAM IS DISMISSED UNTIL INPUT TERMINATES (USER TYPES WAKE-UP CHARACTER)

FIGURE 1. Typical Sequence of Events Elicited by a User's Command

Fig. 2 we have represented events for each of three processes in the balance set. The bottom horizontal line represents time, t, in milliseconds. The user who owns process 1 finishes inputting a command (TIWK) at t=20. This causes the process controller to reassign priorities and the balance set control to estimate storage requirements. The core manager seesthat room is provided in core for the new process and the swapper is activated. The first page of process 1 is brought in and a very short burst of CPU service follows, ended by a page fault. About 20 milliseconds later, the page requested arrives and it so happens that the CPU is available. Process 1 gets another short burst of computation, until it page faults again. Processes 2 and 3 are also in the balance set and the CPU service bursts that they receive are interspersed among those of Process 1. Notice that the fourth burst of Process 1 and all bursts of Process 3 begin considerably later than the moment the page they requested has actually arrived in core. At t=200 milliseconds, Process 2 blocks for I/O. That is, the process stops running because information must be transferred to or from the external world in a slow device; for example, the process waits for the user to type something into his Teletype. At this point, the Process Controller and the Balance Set Controller may decide to bring a different runnable process into the balance set and throw out Process 2. After some time, Process 1 finally blocks, and Process 2 wakes up again.

2.2.4 The Queueing Network Model

From this admittedly sketchy description of the internal workings of TENEX, we may now proceed to present the structure of our model—a state diagram, comprising the network of servers and their attending queues of user programs, that is represented in Fig. 3.

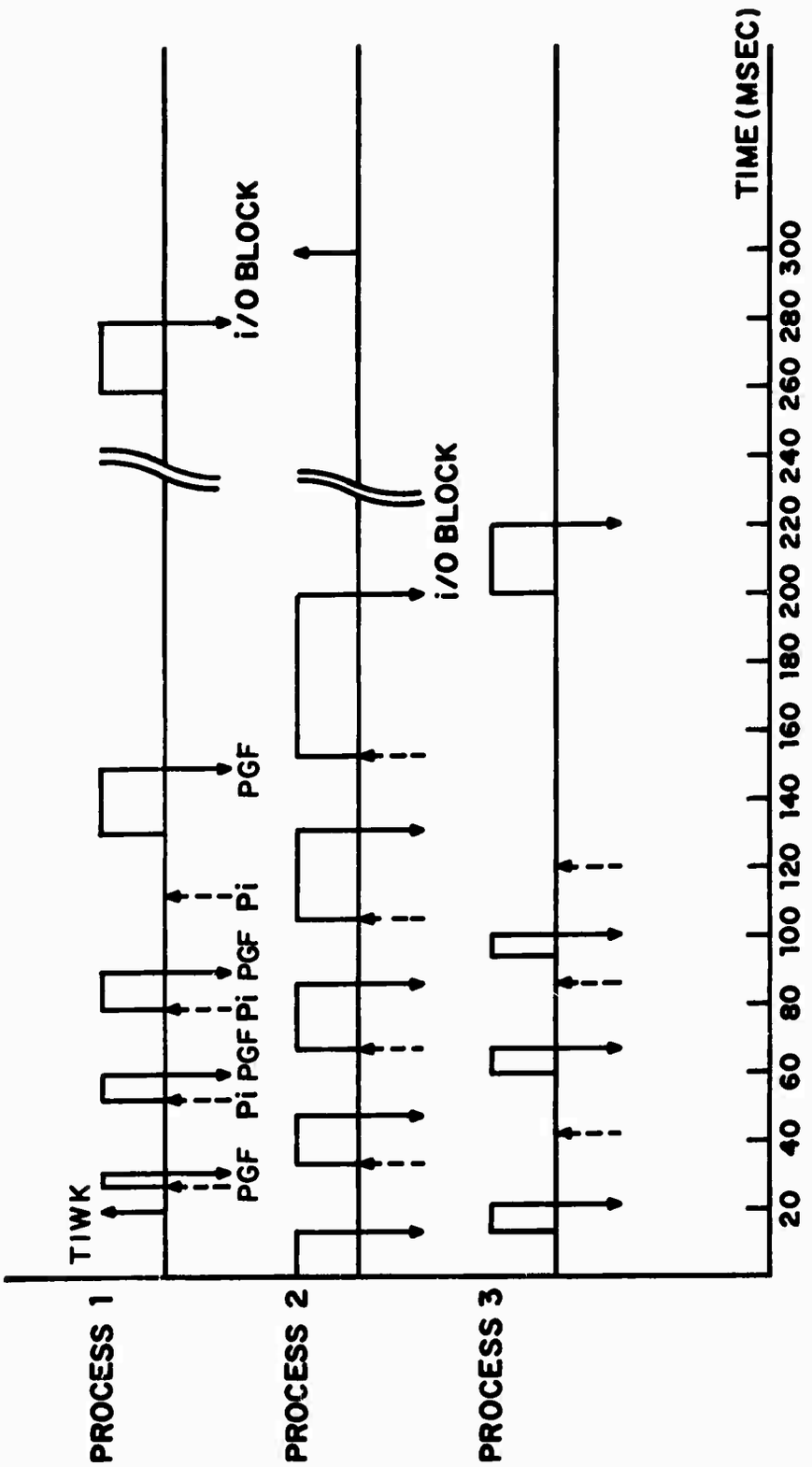


FIGURE 2. Typical Sequence of Events for Three Interactive Users

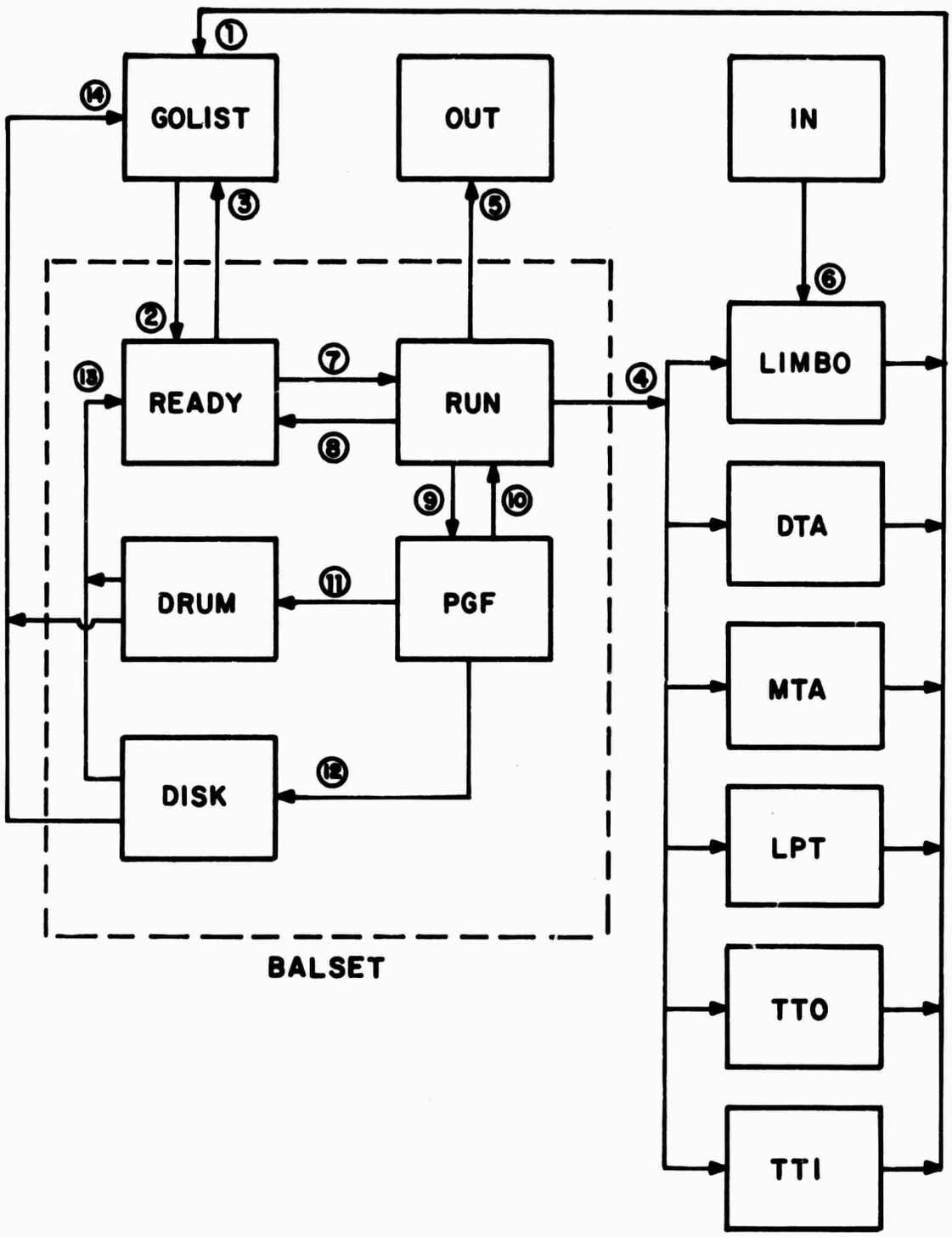


FIGURE 3. Model of the Behavior of Programs in TENEX

All runnable programs are either in GOLIST or in the set of states included in the dashed box called BALSET. Runnable programs are those programs which have completed their I/O and are waiting to be executed (or are being executed). A subset of these, selected by the balance set controller, has had core memory allocated to it and is considered to be compatible (their working sets can all fit together in core, simultaneously). Programs in the balance set can be removed therefrom and placed in GOLIST, and vice versa, depending on their priorities as judged by the balance set controller. Programs in the READY state (those which are both runnable and in the balance set) are selected for execution by the scheduler and enter the RUN server. RUN service is terminated for one of several reasons:

a) The program is I/O blocked, involving service by any of the several services available, such as dectape (DTA), magnetic tape (MTA), lineprinter (LPT), terminal output (TTO), and terminal input (TTI). The box labeled LIMBO corresponds to several instances of suspended animation in which a program may find itself as a consequence of the operation of the pseudo-interrupt system.

b) The program runs for its full quantum and is returned to the READY state. Here, the balance set controller will determine whether the program must be thrown out of the balance set because of demands from the runnable programs (in GOLIST), or whether it can be allowed to stay in READY state.

c) The program may finish computation altogether; i.e., the user logs out (OUT).

d) A page fault has occurred (PGF). In this case either the page referenced is still in core (for example, after a quantum overflow that did not result in the program's being thrown out of the balance set) in which case running is simply interrupted but not finished, or the page referenced must be brought in from DRUM or DISK. After the page has been brought in, the program may go back to READY state, or may find that during the time taken by the page transfer, the balance set controller decided to throw the program out of the balance set.

As we can see, the GOLIST and the READY states of our diagram really correspond to user programs waiting to be processed; i.e., they represent waiting lines. All the other states except IN and OUT represent servers with different characteristics. For example, TTI and TTO can be considered parallel, multichannel servers capable of servicing simultaneously as many programs as there are active terminal lines, while DRUM can serve as many programs in one drum revolution as there are non-superimposed transfer requests (superimposed requests would be those involving overlapping drum azimuths). Others, such as RUN and DISK, must be considered as single channel servers capable of servicing one user program at a time.

In summary, each server is characterized by the way in which waiting programs are selected for service (queue discipline), by the number of programs that can be serviced simultaneously, by the probability density of its service time, and by its transition probabilities (the probabilities with which programs will request their next service to be performed by another server). A measure of these quantitites is all that is required to identify and quan-

titatively define the model. From the model, characteristics such as the number of programs in any of our states, the load factors for each server, the distribution of waiting times - the quantities that are needed to satisfy our goals of description and prediction of system response characteristics - can be obtained.

We believe that the measured service time probabilities, as well as the somewhat "ad hoc" queue disciplines used in TENEX, will preclude the use of analytical results that are available in the Operations Research literature. Therefore, we expect that our final result will be a Monte-Carlo driven simulation model.

2.3 MEASURING SYSTEM

We have designed, implemented, and partially debugged a software measuring system to obtain the statistics we need to specify quantitatively our model. The measuring system consists of two parts: a set of software inserts and a special user program. The software inserts are patched directly into the TENEX Monitor at points corresponding to the numerals in Fig. 3. Every time a user's program is dismissed for I/O, for example, it activates a patch inserted at an appropriate point in that section of the monitor code that performs the dismissal.

The patch gathers data, compacts it into two PDP-10 36-bit words and records it in a buffer located in the monitor's address space. The data gathered are the following:

a) The measurement number (the numeral in the corresponding position in Fig. 3)

b) The job number (identifying the user)

c) The fork number (what process in the hierarchy of processes the user program may have spawned)

d) The time of day (in milliseconds)

e) State dependent data, such as the I/O blocked condition, i.e., what I/O device is involved. These data are specific to the example chosen; for other measuring points they would be irrelevant and other types of data would be substituted (for example, measure 9 will contain the desired initial page number and measures 10, 11, and 12 will give the real core page number assigned.

The special user program has the following functions:

a) It allows the user to specify an I/O device for permanent storage of the measurement data, as well as to write headings and other indexing information.

b) It copies the entire monitor code as the first record of the data. This is done to facilitate reduction of the measurement data, and to help explain possible anomalies in the data produced by undocumented changes in the monitor.

c) It inserts the patches into the monitor code and dismisses itself (goes to LIMBO) until the special wake-up condition described next is met.

d) When the buffer is more than a given percent full, the program wakes up, dumps the contents of the buffer onto the I/O device selected in a), checks whether the user has signaled termination of the measurement, and if he has not, goes back to sleep. This loop is then repeated.

Two data-reduction programs are available to unscramble the data recorded: a time history program and a histogram-generating program. The time-history program will simply translate the bit patterns of the raw data into easily readable descriptions of the event recorded so that the gyrations of any particular program in the time-sharing system can be followed and interpreted. The histogram-generating program will produce and make available the probability densitities that we need for our modelling.

This measuring system and the time-history data reduction program have been written and debugged. Some data have been obtailed already and we expect to be able to gather the bulk of our data in the immediate future.

3. USAGE STATISTICS

3.1 SESSION DURATION AND CPU TIME CONSUMED

Each usage of the time-sharing system by an individual user is called a session. For each session, the TENEX accounting system keeps track of the time elapsed between login and logout, and of several computer resources used during that time interval. As a first attempt at characterizing (and modelling) the behavior of users, we have collected data on the length of sessions of our TENEX time-sharing system, along with the CPU time consumed in each session.

We have examined data for all the usages of the TENEX time-sharing system from 1 December 1970 to 30 June 1971, a total of more than 14,000 sessions. A special feature developed especially for our purposes allows these data to be classified and sorted in a two-dimensional histogram, recording the number of sessions lasting between T_n and T_{n+1} minutes and consuming between C_m and C_{m+1} seconds of CPU time. As a compromise between resolution and size, we adopted a log-linear time scale (giving progressively longer time intervals) according to the formula

$$T_n = 2^{E(n/3)}[15+5(n \bmod 3)]-15 \text{ (in minutes)}$$

where $E(n/3)$ is the greater integer $<n/3$. This gives, for $n = 1,2,3,\ldots7$, the values 5, 10, 15, 25, 35, 45, 65. Exactly the same expression was used for C_m, except that times were expressed in seconds. The measured relative frequencies corresponding to such a histogram are reproduced in Table I. We observe, for example, that 10.37 percent of all sessions recorded here were less than 5 minutes long, and consumed less than 5 seconds of CPU.

T (mins)

C (secs)	5	10	15	25	35	45	65	85	105	145	185	225	305	385	465	>465
5	1037	145	59	43	22	15	19	11	10	11	1	1	3	1	1	5
10	360	237	96	97	63	23	33	17	9	10	4	4	4	1	0	2
15	113	140	94	95	44	27	35	18	9	7	5	5	4	1	1	1
25	85	170	119	155	85	58	74	43	28	24	9	2	3	1	1	3
35	43	73	76	133	87	55	56	40	18	29	11	4	5	1	2	1
45	9	43	49	103	69	37	51	40	24	23	9	3	7	1	0	1
65	10	53	62	96	105	83	98	63	33	36	21	6	4	2	4	1
85	4	16	34	67	58	48	79	49	33	36	24	11	6	5	4	1
105	1	11	15	46	44	39	65	44	36	36	21	10	4	5	1	1
145	2	8	19	55	48	57	110	81	51	54	27	16	14	8	6	2
185	1	6	8	38	33	51	64	56	46	53	29	15	9	5	2	1
225	1	2	1	14	24	16	51	45	45	64	33	17	23	9	6	1
305	1	2	3	19	21	35	61	69	57	80	49	36	31	8	9	4
385	1	1	6	9	11	18	40	40	35	52	33	23	29	13	9	4
465	1	0	0	5	6	11	25	16	22	43	29	17	24	9	7	2
625	1	0	1	3	9	19	42	33	36	57	33	31	39	28	13	7
785	0	0	0	1	4	6	31	19	20	39	28	11	33	12	6	4
945	1	0	0	0	1	6	12	24	15	27	14	13	21	15	4	7
1265	0	0	0	0	1	2	9	21	16	19	28	22	23	16	14	15
1585	0	0	0	0	0	1	1	3	12	15	10	9	16	9	4	9
1905	0	0	0	0	1	0	1	1	4	6	8	6	9	9	5	4
2545	0	0	0	0	0	1	0	1	1	6	8	5	11	9	6	11
3185	0	0	0	0	0	0	0	1	0	1	1	2	7	4	2	8
3825	0	0	0	0	0	0	0	0	1	0	1	2	2	1	1	6
5105	0	0	0	0	0	0	0	0	0	1	1	0	2	1	4	5
6385	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	4
7665	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

TABLE I. (in 10^{-4} units) Joint Relative Frequencies of Session Durations and CPU Consumed

We begin our analysis by computing some statistics. It will be assumed throughout that the relative frequencies in Table I are a discrete probability density function representing events that occur at the arithmetic mean of the interval. For example, all sessions lasting less than five minutes and consuming less than five seconds of CPU will be represented by a session of 2.5 minutes duration and consuming 2.5 seconds of CPU. We shall also adopt the following terminology:

$P(T_n, C_m)$ is the probability density of a session lasting between T_{n-1} and T_n minutes, consumbetween C_{m-1} and C_m seconds of CPU.

$P(T_n) = \sum_{m=1}^{28} P(T_n, C_m)$ and

$P(C_m) = \sum_{n=1}^{16} P(T_n, C_m)$ are the marginal probability densities of session duration and CPU consumed, respectively.

$P(T_n|C_m) = P(T_n, C_m)/P(C_m)$ and

$P(C_m|T_n) = P(T_n, C_m)/P(T_n)$ are the conditional probability densities of session duration given CPU consumed, and of CPU consumed given session duration, respectively.

$$E[g(T)] = \sum_{m=1}^{28} \sum_{n=1}^{16} g\left(\frac{T_n + T_{n+1}}{2}\right) P(T_n, C_m)$$

is the expected value of the function g(T) of session duration. Similar definitions apply with respect to CPU consumed and with respect to the conditional and the marginal densities.

$$\rho = \frac{E[(T-E[T])*(C-E[C])]}{\sigma_T * \sigma_C}$$

is the correlation coefficient between session duration and CPU consumed, where

$$\sigma_T = SQRT(E[T^2] - E^2[T]) \text{ and}$$
$$\sigma_C = SQRT(E[C^2] - E^2[C])$$

We present in Table II the means and standard deviations of both conditional and both marginal densities. Inspection of Table I shows that a strong correlation exists between session duration and CPU consumed. However, the relatively low value of the correlation coefficient (see Table II) shows that this correlation is not linear. An excellent linear fit is obtained by computing E[log T|C] and E[log C|T] and plotting it on semi-log paper (see Figure 4). One gets:

$$E[\log C|T] = .20 + \log T$$
$$E[\log T|C] = .42 + 0.66 \log C$$

These expressions are especially suited to our modelling work because they allow us to estimate, for example, how much CPU will be consumed, on the average, in a session of duration T.

Let us now turn our attention to the different probability densities involved. Figures 5 and 6 show plots of the cumulative conditional probabilities

$$Pr[T<T_N|C] = \sum_{n=1}^{N} P(T_n|C) \text{ and}$$

$$Pr[C<C_m|T] = \sum_{m=1}^{m} P(C_m|T)$$

TABLE II

CONDITIONAL AND MARGINAL STATISTICS

T(mins) or C(secs)	Session Duration			CPU Consumed		
	E[T|C]	$\sigma_{T\|C}$	Weight	[E[C|T]	$\sigma_{C\|T}$	Weight
5	11.1	38.9	.138	8.3	29.2	.167
10	18.7	38.7	.097	21.6	29.4	.091
15	27.3	48.8	.060	36.0	49.2	.064
25	33.4	46.7	.086	57.1	72.5	.098
35	41.1	51.5	.063	83.6	123.2	.074
45	47.5	52.9	.047	132.5	188.8	.062
65	53.7	58.1	.068	174.9	217.4	.096
85	70.0	71.2	.048	228.5	289.4	.073
105	74.3	66.5	.038	287.3	351.3	.056
145	83.3	75.9	.056	342.6	405.2	.073
185	86.2	68.4	.042	449.6	525.4	.044
225	117.7	87.0	.035	528.3	574.5	.028
305	121.1	89.2	.048	728.1	863.8	.034
385	137.6	100.4	.033	830.2	855.3	.017
465	149.0	100.2	.022	899.2	1095.5	.011
625	157.8	109.6	.035	1766.6	1922.6	.012
785	162.2	106.1	.022			
945	176.0	116.6	.016			
1265	215.2	127.4	.019			
1585	224.4	123.2	.009			
1905	244.1	120.1	.005			
2545	294.0	125.2	.006			
3185	325.8	117.1	.003			
3825	333.5	127.3	.001			
5105	376.6	106.1	.001			
6385	420.6	66.5	.001			
7665	265.0	.0	.000			

E[T] = 72 min E[C] = 205 secs

σ_T = 93 min σ_C = 472 secs

p = 0.55

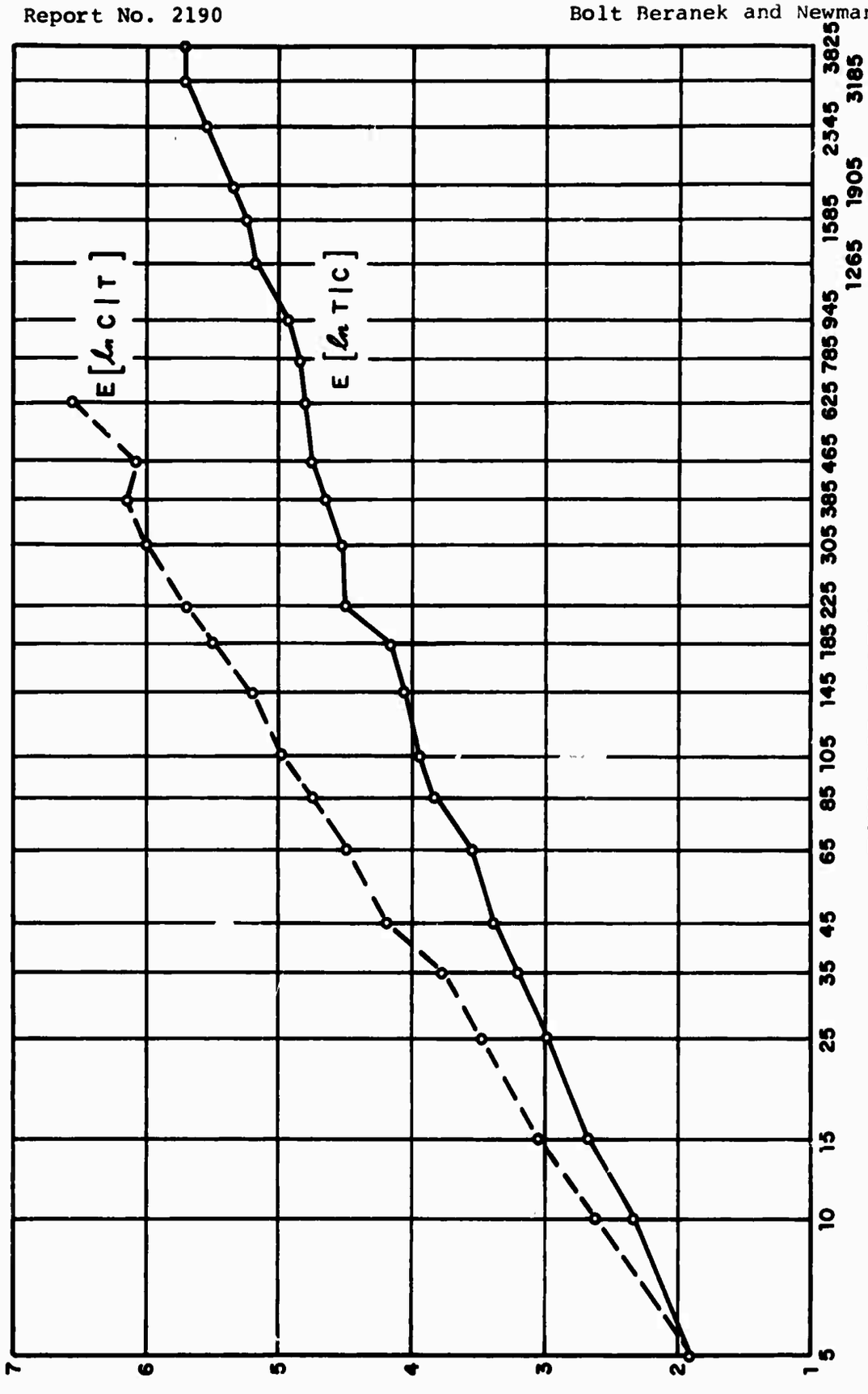


FIGURE 4. Conditional Expected Values of the Natural Logs of Session Duration and of CPU Consumed

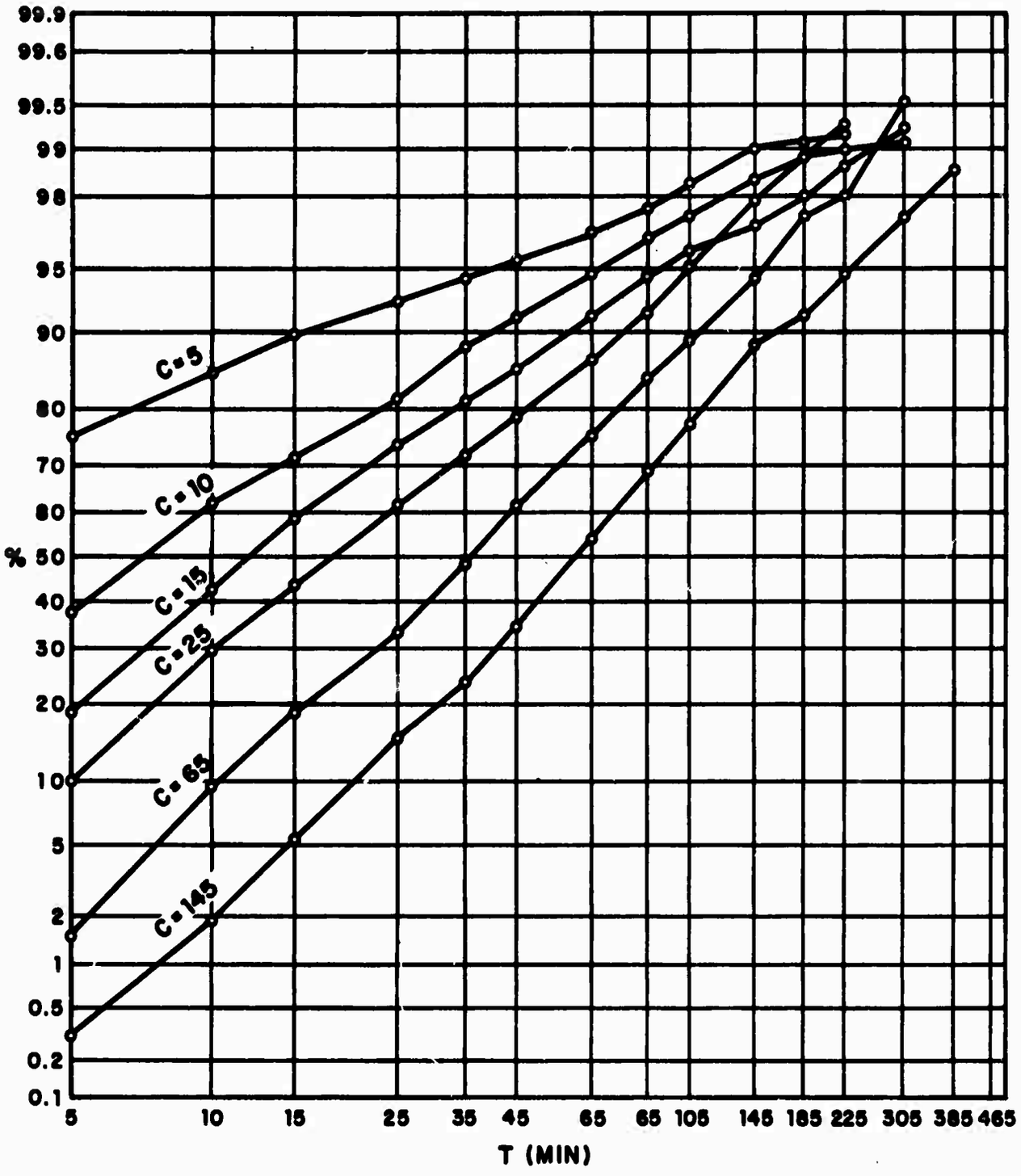


FIGURE 5. Cumulative P(T/C)

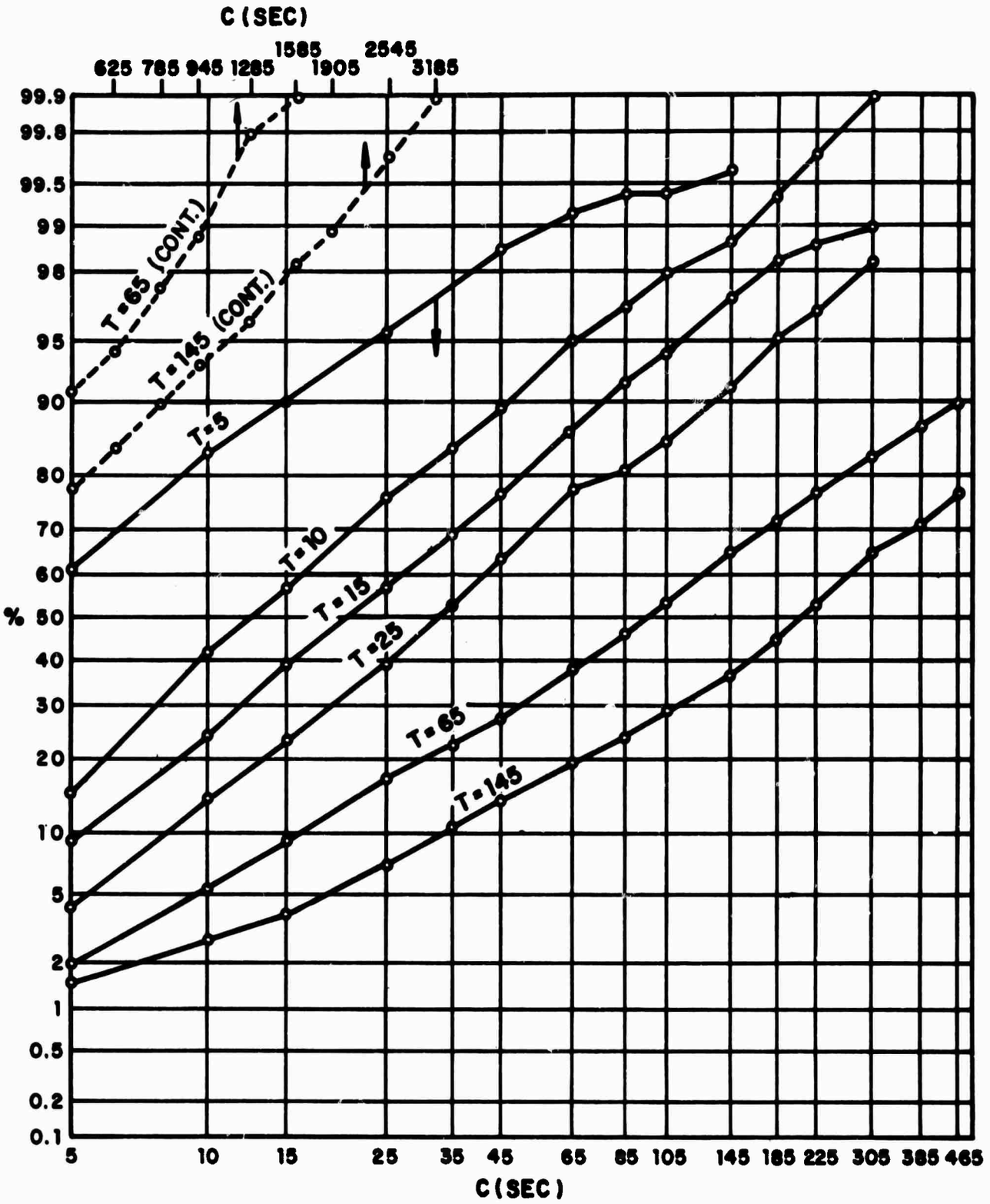


FIGURE 6. Cumulative P(C/T)

on logarithmic normal probability paper. The fit to a log normal distribution is good for P(T|C) and not quite so good for P(C|T).

Plots of the marginal densities P(T) and P(C) are presented if fig. 7. While the hypothesis of lognormality could be defended for P(C) it appears to be untenable for P(T). We are now attempting to find other distributions that fit the data.

3.2 SYSTEM STATISTICS

One of the features of the TENEX Executive System allows certain privileged users to obtain information related to the performance of the time-sharing system. This facility, called STATISTICS, provides the following types of information:

1) Allocation of system resources, such as the fraction of the total up-time spent:

   a) running user's program
   b) idling, that is, without any runnable user program
   c) waiting for secondary storage transfers (all runnable user programs have page-faulted)
   d) managing core
   e) handling page faults (included in item a above)

2) The total number of pages read/written from/onto the drum and the disk

3) The amount of core memory available to users

4) The number of times user programs have been dismissed because of terminal I/O, and have been interrupted from the terminal

5) The time integral (in milliseconds) of the number of runnable user programs the system thinks can be simultaneously kept in core.

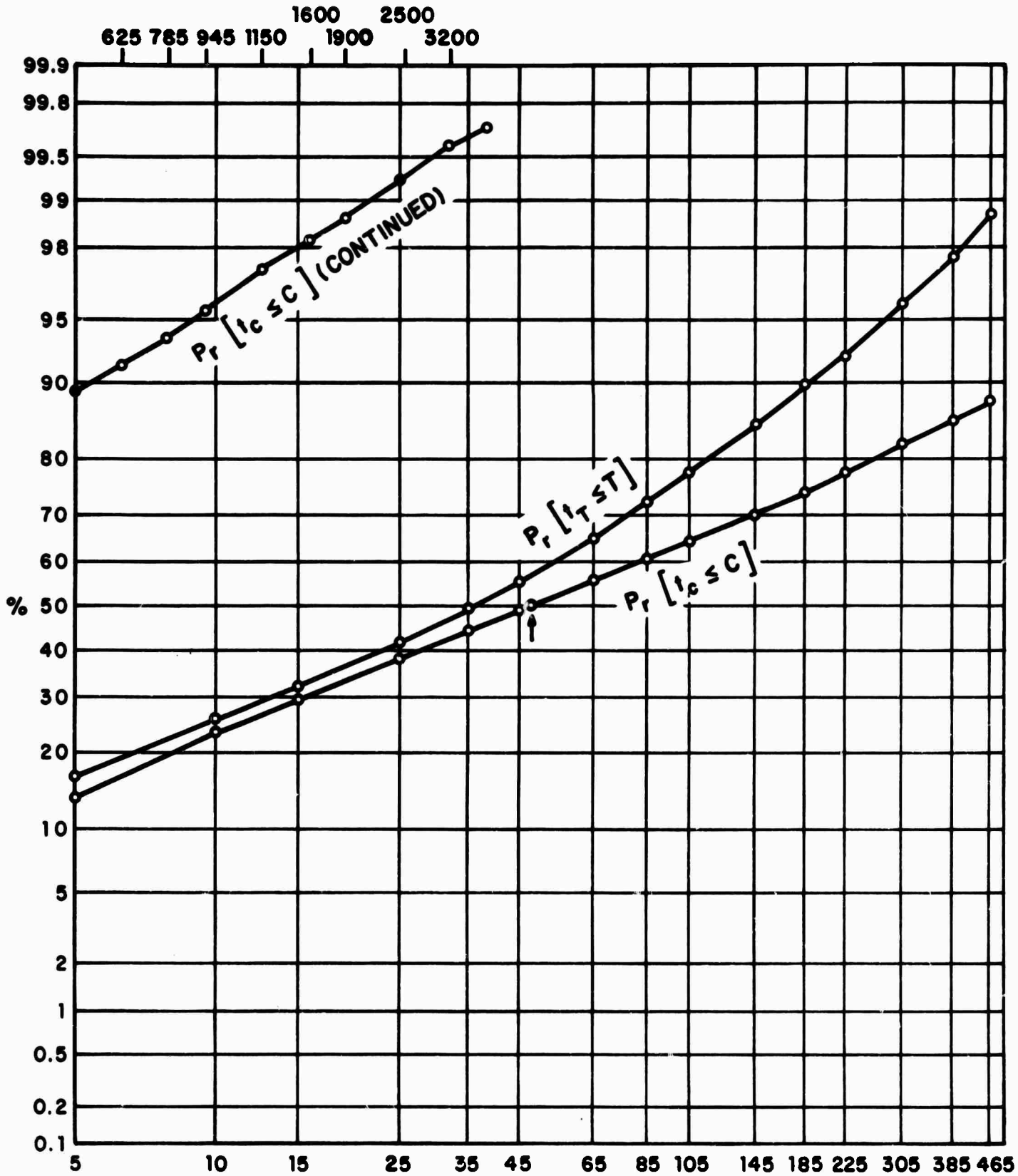


FIGURE 7. Cumulative P(T) and P(C)

6) The running time (in milliseconds) of user programs in each of the five queues of the system.

7) Allocation of subsystem usage. For each subsystem available in TENEX (LISP, TELCOMP, TECO, EXEC) the following information is provided:

a) total time spent running user program
b) number of page faults

We have gathered data by running STATISTICS at 0900 hrs. and at 1800 hrs. for 48 consecutive working days comprising the entire months of February and March and part of April 1971. Fig. 8 shows a typical printout of these data. We have processed these data and we shall proceed now to report some of the results that are of interest.

The average UP time as measured at 1800 hours was 14 hr., 30 min. The average time spent running user programs was 242 minutes or almost exactly 4 hours. This was obtained by averaging the result of subtracting idling, waiting, and core managing time from UP time.

In Table III, we present some statistics obtained by analyzing the afternoon data. Averages and standard deviations were obtained by weighting the corresponding quotients in proportion to the day's running time. Thus, for example, for the first entry in the table, the formulas used were

$$AVE = \sum_{n=1}^{48} TWK_n/SUMRUN,$$ where TWK is number of Terminal Wake-Ups

```
!DAYTIME
 TUESDAY, APRIL 13, 1971 18:00:27
!SYSTAT

 UP 25:41:39  13 JOBS

 JOB TTY  USER       SUBSYS
 2   57   SATTERFIELD EXEC
 3   46   WEBBER     (PRIV)
 4   40   JSIEGEL    EXEC
 6   21   GRIGNETTI  EXEC
 7   31   MURPHY     MACRO
 9   27   WEGBREIT   TECO
 12  DET  HANSON     (PRIV)
 13  37   REMINGTON  TELCOM
 15  35   HARTLEY    LISP
 16  34   WOODS      EXEC
 19  56   CALLEVA    EXEC
 23  60   MURPHY     EXEC
 24  17   HOLLISTER  MDEVEL

!STATISTICS

 IDLE 34%  WAITING 29%  CORE MGMT 3%  PAGER TRAPS 8%
 DRM READS 1999463 WRITES 998211  DSK READS 126238 WRITES 49453
 83 PAGES OF USER CORE
 67640 TERM WAKEUPS  3283 TERM INTERRUPTS
 TIME INTEGRAL OF # JOBS IN BALANCE SET 123201023
 RUNTIME OF JOBS ON Q'S 0-4 (MSEC)
        :510700 3757137 6641200 3318943 16149699

 SUBSYS     TIME     PAGE FAULTS
 EXEC      6118313     357235
 (PRIV)   12021601     739050
 NOTIFY      18800        325
 MDEVEL    2196129     159169
 TECO      1503492      64478
 LOGO       322487      34916
 TDEVEL     859328      70704
 BASIC        5392        332
 PARSEC    1286714     150693
 F40         43765       2991
 LOADER     329226      21485
 SETMRP        370         55
 TLINK        4255        340
 LISP      4203344     360087
 RUNOFF     413529       4770
 DISCUS        866         81
 MACRO      496650      13097
 TELCOM    1007414      47696
 CRKPNT      23528       1467
 ULIST       25226        566
 DELD        27930       1379
 SRCCOM      95805       1337
 WATCH       11334        766
 SNDMSG      11944       1063
 SDDT          929        101
```

FIGURE 8. Statistics Print-out

TABLE III

SYSTEM STATISTICS

Description	Ave.	Std. Dev.	Units
Terminal wakeups	140	30	per min run'g
Waiting	.66	.21	mins/min run'g
Managing Core	.085	.03	mins/min run'g
Handling Page Traps	.19	.06	mins/min run'g
User Programs in Core	3.3	0.97	
Drum Reads and Writes	4540	1210	pages/min run'g
Drum Reads	2933	914	pages/min run'g
Drum and Disk Reads	3116	922	pages/min run'g

$$ST.DV.=\left[\left(\sum_{n=1}^{48} TWK_n^2/RUN'G_n\right)/SUMRUN-AVE^2\right]^{1/2}$$

where

$$SUMRUN=\sum_{n=1}^{48} RUN'G_n$$

It should be realized that the value given in the table is not the standard deviation of the number of terminal wake-ups as they would be counted at the end of each minute of running time. It is instead the standard deviation of a set of 48 large sample averages obtained as ratios of a large number of terminal wake-ups to a large number of minutes of running time. When this fact is considered, the standard deviations appear to be rather large. An estimate of the true standard deviation can be obtained by multiplying the standard deviation of the sample averages by the square root of the average running time, 242 minutes. So the standard deviations per minute of running time appear to be 15.6 times as big as the ones in Table III. Apparently, we are dealing either with highly skewed distributions with very long tails, or with multimodal distributions. When we obtain data with our complete measurement system, the forms of these distributions will become clear.

A better understanding of these data can be gained by plotting the various items whose descriptions appear in Table III, versus running time and versus the number of page faults. These plots show a high linear dependence, and computation of the best linear fits produces the results detailed in Table IV.

Thus, for example, running time (RT) appears to be a good predictor of waiting time (WT)—the prediction equation being:

$$WT = 3.76 + .67*RT$$

with a correlation coefficient (ρ) of .96 and with a root mean square error of 33.2 minutes. Notice that since the intercept is small, the slope of the prediction line coincides with the average value of the ratio WT/RT given in Table III, as it should.

TABLE IV

Linear Regression Statistics		Running Time (mins)	Page Faults (thousands)
Waiting Time (mins)	Slope	.67	.22
	Intercept	3.76	3.9
	ρ	.96	.91
	rms error	33.2	47.7
Core Management Time (mins)	Slope	.069	.025
	Intercept	2.74	1.22
	ρ	.83	.87
	rms error	7.36	6.55
Trap Time (mins)	Slope	.159	.061
	Intercept	5.68	.51
	ρ	.86	.97
	rms error	14.9	7.7
Page Faults (thousands)	Slope	2.55	
	Intercept	117.5	
	ρ	.87	
	rms error	231.4	
Time Integral of Jobs in Core (mins)	Slope	2.89	1.00
	Intercept	116.3	74.4
	ρ	.95	.96
	rms error	154.9	137.4

Notice also the strong correlation that exists between running time and page faults. This strong correlation makes it undesirable to attempt fitting the various items by means of double regression (on both running time and page faults, simultaneously). In fact, if we did so, we would find that the variability of the result would be very large, due to the smallness of the determinant of the covariance matrix. It is better and simpler to divide the various items by the running times, and to fit the quotients to the number of page faults per unit running time. The results of this appear in Table V.

3.3 SUBSYSTEM STATISTICS

In this section, we present statistics corresponding to selected subsystems of the TENEX time-sharing system. A subsystem is defined in TENEX as any executable program that is stored in the SUBSYSTEM directory. The range of usage of the subsystems varies considerably. One, the EXECUTIVE language, is used by all the TENEX users, since it is the handle with which they communicate and work with TENEX. Others, such as LISP, FORTRAN, TECO (an editing language), TELCOMP (a conversational language), etc., are of common and widespread use, but many others are private programs that may be actually executed by a single user. We shall concentrate here on a few subsystems of general interest, for which sufficient use has been observed to make our data reliable.

Statistics on the allocation of running time among subsystems and on their page faulting rate are presented in Table VI. It is informative to notice that EXECUTIVE usage represents close to one-fifth of the total running time——a result that is somewhat surprising, since usage of the EXECUTIVE would appear subjectively to most users as being much less.

TABLE V

		Page Faults (thousands per min. running time)
Waiting (per min. running)	Slope	.063
	Intercept	.495
	ρ	.40
	rms error	.35
Core Management (per min. running)	Slope	.019
	Intercept	.023
	ρ	.60
	rms error	.024
Trap (per min. running)	Slope	.049
	Intercept	.033
	ρ	.82
	rms error	.032
Jobs in Core (per min. running)	Slope	.56
	Intercept	1.69
	ρ	.81
	rms error	.39

TABLE VI

SUBSYSTEM USAGE STATISTICS

SUBSYS	% of Total Running		PG FLTS per Running Second		% of Total PG FLTS	
	Ave	St.Dev.	Ave	St.Dev.	Ave	St.Dev.
Executive	18.5	5.0	52.3	9.6	18.6	9.5
Private Progs	25.0	7.1	55.5	19.2	26.7	13.7
Lisp	12.1	8.7	63.0	22.4	14.6	9.6
Telcomp	2.7	3.1	41.6	30.4	2.1	4.0
Fortran IV	1.0	.8	23.9	9.2	.5	.7
MACRO	3.3	3.1	18.1	9.2	1.1	2.1
Loader	5.0	2.8	24.4	10.3	2.3	1.3
Teco	7.4	4.6	24.1	12.4	3.5	3.5
Runoff	.8	.9	9.2	5.4	.1	.2

Under the category PRIVATE PROGS, we have included user programs that are run as private programs, that is, executable programs that are not stored in the SUBSYSTEM directory. Observe also that the average percentages do not add up to 100—a consequence of the fact that only a few of the available subsystems are treated here.

As with the system statistics, a better insight can be gained by plotting page faults versus running time, and, again, after noticing the linearity of the scatter diagram, by computing the best linear fits. The results are shown in Table VII.

A summary of these results that should be useful for system designers and operators can be obtained by classifying the various subsystems (including some that were not included in Table VI) into the categories of Table VIII.

3.4 CONCLUSIONS

The significance of these results will become clear in Section 4. As discussed in that section, our entire approach rests on characterizing interactions in terms of three basic computer resources demanded by the users' commands—running time (CPU), core required, and amount of I/O. Since with a given computer's configuration, the number of page faults is a function of the core required, the results presented in Section 2.3 show the manner in which running time and core required are statistically related, and show that such relationships are a function of the subsystem being used. Also, the results presented in Section 2.2 allow us to predict waiting time, core management time, etc., as a function of running time. This is

Table VII

BEST LINEAR FITS OF SUBSYSTEM PAGE FAULTS (THOUSANDS) ON SUBSYSTEM RUNNING TIME (SECONDS)

SUBSYSTEM	SLOPE	INTRCPT.	CORREL. COEFF.	RMS ERROR
EXECUTIVE	.049	7.96	.97	22.0
PRIVATE PROGS.	.049	25.06	.91	69.7
LISP	.053	18.11	.96	34.8
TELCOMP	.032	3.61	.83	12.2
FORTRAN IV	.020	.60	.91	1.49
MACRO	.017	.53	.86	4.84
LOADER	.020	3.38	.87	7.17
TECO	.011	14.5	.59	13.3
RUNOFF	.009	.051	.83	.85

Table VIII

ALLOCATION OF RUNNING TIME AND PAGE FAULTS
TO THE PRINCIPAL USER FUNCTIONS

FUNCTION	SUBSYSTEM Included	% Total Run'g Time	% Total Page Flts.
Communicating with TENEX	EXEC.	18.5	18.6
Editing	TECO SRCCMP RUNOFF	9.3	3.8
Compiling of Loading	TECO MACRO LOADER	9.3	3.9
Interactive Languages	TELCOMP BASIC	3.9	2.2
LISP	LISP	12.5	15.3
Special purpose	PRIV	40.7	49.4

important because it contributes towards allowing us to predict the computer's response time as a function of the resources demanded.

4. USER MODELS

The gathering of daily statistics of time-sharing system performance through the use of the measurement system described in section 2 will enable us to understand how the system behaves over a period of time. To understand why it behaves this way and how it might behave under different conditions, we require models of user behavior in addition to models of time-sharing system behavior.

4.1 INTRODUCTION

During the past few months, our views concerning the structure of user models have evolved considerably. In our previously reported work, we concentrated on building an understanding of users' problem-solving strategies and of the fine structural details of their command-selection procedures. We attempted to account for why a user chose a particular command at a particular time. The approach required carefully controlled experiments in highly constrained situations in order to delimit the options among which the users could chose. Our MINITECO text-editing experiments constituted an example of this technique.*

Gradually, we came to the realization that building models at this level is an impossibly slow process, because the results are highly dependent upon the task being studied and upon the constraints imposed. We turned to less constrained, more realistic tasks, such as FORTRAN debugging, and redefined our goals. We decided to settle for a statistical description of the commands chosen and turned toward Markov models of user behavior. We

*Semiannual Report #7, July 31, 1970, ARPA Order #890, Amendment 4.

conducted preliminarly experiments with FORTRAN debugging tasks and took a hard look at what kinds of information could be extracted from them.

While we were reexamining our approach to user modeling, we were also making rapid progress in formalizing our time-sharing system models. As this occurred, we could begin to assess how the user and computer models would have to interface and whether the user models being contemplated would yield the outputs required by the computer models. We have now concluded that the command-choice models previously discussed are simply not appropriate for our purposes.

One difficulty with command-choice models is the large number of them that would be required to treat the wide variety of users and tasks represented on a multi-purpose time-sharing system. Another difficulty is that there is poor correspondence between the type of command chosen by a user and the actual computational load placed on the time-sharing system. There are two principal reasons for this:

(1) the computer resources demanded by a particular command are highly context-dependent; it makes no sense, for example, to speak of the resources demanded by a COMPILE command without specifying at least the size of the file being compiled; and

(2) the fundamental unit of interaction between the user and the computer is not really the command; in many circumstances, commands are concatenated and processed in a single interaction, while in other cases a single command may give rise to a whole series of interactions as the computer requests several items of information from the user.

We have now concluded that our user models must be structured around the basic user-computer interaction cycle and must yield outputs in terms that are relevant to the computer models, namely, the amounts of various computer resources being demanded during a particular interaction.

We see the development of these user models as a three-stage process. The first stage involves finding descriptors for user demands that are general enough to encompass widely different classes of users who are using the time-sharing system in quite dissimilar ways. The second stage involves validating these descriptors and demonstrating that they are sufficiently stable to characterize adequately the behavior of a specific class of users over some period of time. The third stage involves the development of mathematical techniques for describing the manner in which the descriptors change in response to changes in the computer system characteristics.

The remainder of this section elaborates our plans.

4.2 DESCRIPTORS FOR USER DEMAND

In Section 2.2, we discussed the various events that can occur during an interaction cycle. Of these, we chose the TIBLK (where the program becomes blocked while waiting for additional teletype input from the user) as a salient point marking the beginning of an interaction cycle. We defined the time between TIBLK and the next TIWK (the teletype input wake-up caused by typing the terminating character of a new command string) as the user response time (URT). We defined the remaining part of the cycle—the time between a TIWK and the next TIBLK—as the computer response time (CRT). (See Fig. 1 for a graphic representation of these parts of the interaction cycle.)

The CRT for an interaction is a function of the computer resources demanded by the user during that interaction. Specifically, we have identified three important system resources by which such a demand may be characterized.

x_1 = CPU time
x_2 = core
x_3 = input/output

For notational purposes, we define the vector

$$\underline{x}_i = (x_1, x_2, x_3)_i$$

as the user demand during interaction i.

Our object is to describe, in some statistical manner, the user interaction characteristics, URT and $\underline{x}$. We, therefore, seek the joint probability density $p(\underline{x}_i)$ of the resource demand. In addition, we must describe the temporal characteristics of a series of demands, i.e., the probability density function for URT_i.

We expect that URT will depend strongly on other interaction descriptors; namely, the resources demanded. Therefore, we have decided to obtain the conditional probability density function

$$p(URT_i | \underline{x}_{i-1}, \underline{x}_i)$$

where $\underline{x}_{i-1}$ and $\underline{x}_i$ are, respectively, the resources demanded in the last interaction and the resources to be demanded in the present interaction. The probability density for URT alone can be obtained, if desired, by summing over $\underline{x}_{i-1}$, $\underline{x}_i$.

Our motivation for conditioning URT_i in this manner is based, in part, on the following:

a) The resources demanded in the previous interaction, $\underline{x}_{i-1}$, constitute a measure of the interaction complexity. The user will spend some time thinking about the results of the previous interaction. The time he spends will depend, to some extent, on the complexity of the previous interaction, especially on the amount of output generated, x_3. In addition, URT_i will depend on the CRT_{i-1} (which, in turn, should correlate highly with $\underline{x}_{i-1}$). The user might in part plan his next request while awaiting the results of the last interaction. This would have the effect of shortening URT_i.

b) is the computer resource that is about to be demanded by the user. We expect a substantial correlation between the time spent planning a demand, URT_i, and the resources demanded, ($_i$).

In summary, we believe that the important aspects of a series of user demands can be characterized by a joint probability density function for resources demanded per interaction and by a conditional probability density function for URT. Of course, certain components of $\underline{x}_{i-1}$, $\underline{x}_i$ may correlate poorly with URT. In this case, the conditional probability density function can be simplified by neglecting these components.

4.3 VALIDATION OF DESCRIPTORS

We are presently implementing data collection mechanisms that will record the required information (CPU, core, I/O, user response time, and computer response time) for each interaction on a user-by-user basis. With this measurement system, we can gather data on individual users or groups of users, either during their normal day-to-day activities or while they are working on specific tasks that we select.

Before we can attempt to model user behavior (i.e., to predict how the user related probability densities change under various circumstances), we must first demonstrate that the descriptors chosen are both general and stationary.

4.3.1 Generality

Each individual user tends to interact with a computer in a unique manner. Studying individual reactions, however, is undesirable (and virtually hopeless). We expect that by measuring the demands of a large number of users over some period of time, we will be able to demonstrate the existence of a relatively small number of user *classes*. These classes would be task-defined, not user-defined. Practically speaking, the

classes should correspond with subsystems available on the time-sharing system, although it may prove necessary, for example, to distinguish between FORTRAN users who are debugging new programs and those who are running existing programs.

The users within a given class would tend to interact with the machine in a similar manner. Each class could, therefore, be characterized by its own unique set of descriptors—presumably, the probabilistic description of LISP users will be different from that of TECO (editor) users. We plan to ascertain whether the descriptors $p(\underline{x}_i)$ and $p(URT_i | \underline{x}_i, \underline{x}_{i-1})$ do indeed characterize the demands of any given user class.

We feel that the identification and description of the user classes would represent a highly useful achievement, independent of subsequent successes in modeling the details of the class behavior.

4.3.2 Stationarity

Parameters that serve to describe the user probability density functions† should be stable for a given class of users when calculated from data collected over reasonably short periods of time. Thus, descriptor parameters calculated for TECO users this week should be reasonably similar to those calculated for this same class last week. We expect that individual differences between users and the jobs on which they are working will be great enough so that for small samples of data (say, 100

†e.g., the moments of the distribution, functional characterizers, etc. As an example, the mean and variance suffice to describe a gaussian distribution.

consecutive interactions) the calculated parameters will show substantial variability. We hope that for larger samples (say, 1000 interactions) enough different users and jobs will be represented in the samples to reduce this variability. If we find that very large samples (say, 10,000 interactions) are required to achieve repeatable results, then the usefulness of the descriptors will be quite limited—data would have to be collected over a period of many weeks or months—and the descriptors generated from this data would not account for short-term variations in user demands. However, this negative result would be in itself, an important conclusion.

To estimate the stability of our descriptor parameters, we plan to proceed as follows: If we have data for 5000 consecutive interactions, we can produce density histograms for resources demanded for the first 100 interactions, the second 100 interactions, etc., and then run Chi-square tests on the hypothesis that all 50 such histograms are drawn from the same population. If we must reject this hypothesis, then we can repeat the calculation for histograms containing 200 or 500 or 1000 interactions, proceeding to pool larger numbers of interactions until we are unable to reject our hypothesis. The smaller the number of interactions, the more stable our descriptors for that sample can be said to be.

Obviously, it is not realistic to pretend that there is some particular sample size for which the descriptors suddenly become stable, where they were not before. We view this tentative procedure, rather, as a consistent way of comparing the relative stability of data obtained under different conditions.

From a mathematical viewpoint, demonstrating the stationarity of user descriptors implies that the density functions $p(\underline{x}_i)$ and $p(CRT_i)$ are not *explicitly* dependent on i. Thus,

$$p(\underline{x}_i) = p(\underline{x}_{i+1}) \stackrel{\Delta}{=} p(\underline{x})$$

for all i, and $p(CRT_i)$ is independent of the specific interaction number; i.e., we have stationarity. (There are some subtle points here regarding the ergodicity of the interaction process. However, they are beyond the present scope.)

4.4 SYSTEM MEASUREMENTS WITH SIMULATED USERS

Once we have identified various classes of users and have characterized their demands, we can begin to make more effective use of our measuring system (described in section 2.3). With real user data, measurements of times spent by each job in each system state, transition probabilities between states, and so forth, will be corrupted by variations in user population and in the types of jobs being run. Thus, whatever is extracted from these measurements is confounded with the effects of a constantly fluctuating load of users working on a large variety of tasks. To explore the interplay between man and machine as a basis for analytic modelling efforts, we must have the ability to perform carefully controlled experiments that are not subject to extraneous variability. However, controlling the real users' demands in the working environment is out of the question. We, therefore, propose to make measurements on classes of "simulated users" whose demands we *can* control explicitly. On a class basis, these simulated users must behave like real users in all statistical respects. This implies that when the real

users are replaced by a set of equivalent simulated users on the time-sharing system, no changes should result in the system measurements obtained. Our procedure is outlined below.

A simulated user in class M, for example, will be designed to generate demands statistically, equivalent to those *measured* for class-M users. We will have characterized these interaction demands in terms of the probability density functions $p(\underline{x})$ and p(URT), so that generating representative demands should be straightforward. Next, the simulated users must be validated by placing them on the time-sharing system and comparing the statistics gathered by our measuring system with the statistics that correspond to real users. If the simulated users do, in fact, mimic real users in all important respects, the results should be indistinguishable.

There is a great potential in having the ability to simulate the demands of "typical" users of various classes. By controlling the user demands over some time period, we can isolate the effects of these demands on the behavior of the time-shared computer system. For example, we can conduct system measurements with controlled numbers of simulated users belonging to a given class, in order to determine how system behavior is affected by changes in user descriptors and in numbers of users. We can also combine different types of simulated users, to study how differing demands may interfere within the computer system. Besides studying the effects of changes in *user* parameters, we can also make certain changes in the *system* (e.g., changes in core allocation or in scheduling algorithms), to determine how system behavior is affected for a selected group of simulated users.

Thus, the two-pronged objective of our experiments with simulated users is to study the *sensitivity* of system behavior with respect to changes in user demand descriptor parameters *and* with respect to changes in computer parameters. The simulated users give us the capability to assess the effects of proposed system changes, assuming that user demand descriptors do not change. We will also have some idea of *how much* these descriptors would have to change in order to produce a noticeable effect on the predicted system behavior. However, the models obtained will *not* account for the changes in *user* behavior that may result from a change in *system* behavior. The simulated users are valid only for the system on which the original measurements were made. Therefore, our next task will be to determine how the user descriptors are likely to change in response to a given change in system behavior. We would then be able to describe completely the overall *closed-loop* man-computer-man-response.

4.5 MEASURING USER BEHAVIOR ON SIMULATED SYSTEMS

To study changes in human behavior that are effected by changes in computer characteristics, we must experiment with real users. However, a major difficulty with such experiments will be to segregate *changes* in user behavior caused by changes in system response from the inherent *variability* in the demands of different users working on different tasks.

We plan to alleviate this difficulty by creating an "adjustable system." The time-sharing system monitor will be programmed to delay system responses to the inputs of any particular user in such a way as to simulate the way the system

would respond with some specified set of system parameters and user demand descriptors. Using this adjustable system, we can isolate any user or group of users from the spurious effects of other users' demands, thus reducing measurement uncertainty arising from human variability.

For a given task, we plan to study how a user's demand descriptors change as we make changes in the simulated system. The words "for a given task" are critical here; for some tasks, a user may have substantial latitude in choosing a strategy of attack, while for others his choices may be quite limited. We plan to derive user demand descriptor parameters for users working on similar tasks under various simulated system conditions. The conditions used will be chosen on the basis of the results obtained from our system measurements with simulated users; we will choose sets of system parameters that produce substantially different system responses to a given set of user demand descriptors, thereby providing maximal incentives for the users to change their interaction strategies.

4.6 ANALYTIC MODELLING OF USER BEHAVIOR

We expect that the outcomes of the preceding series of experiments will provide direction to our analytic modelling effort, in addition to providing valuable data points useful in subsequent model validation. In forming behavioral models for users, we will focus on the modelling and prediction of *changes* in user behavior that arise in response to system changes. We do not expect to be able to construct absolute models of user behavior that are independent of a knowledge of the current operating state of the system. We hope to describe how the measured user probability density functions change as computer parameters are changed.

Our user models, as currently envisioned, will consist of rules for transforming an initial set of user descriptors to a new set for given changes in system response. We can view these transformation rules as a mathematical operation

$$F = \phi(I;\ P_F, P_I)$$

where

I = the initial set of user descriptors
P_I = computer parameters associated with condition I
P_F = computer parameters associated with the new condition F
F = the final set of user descriptors
ϕ = transformation rules that change I into F.

Note that the transformation ϕ depends parametrically on changes in the computer parameters. If these changes are zero, then $P_F = P_I$ and

$$F = I = \phi(I;\ P_I,\ P_I).$$

There are two other properties that the transformation ϕ must possess. They are

(1) Transivity - If a user descriptor changes from d_0 in system condition 0 to d_1 in system condition 1 according to the relation

$$d_1 = \phi(d_0;\ p_1, p_0)$$

then when the system is changed from condition 1 to condition 2, the relation

$$d_2 = \phi(d_1; p_2, p_1) = \phi(\phi(d_0; p_1, p_0), p_2, p_1) = \phi(d_0; p_2, p_0)$$

must hold for any d_1 and p_1. Condition 1 serves as an intermediate state. Thus, $\phi(d_0; p_2,p_0)$ must be the *composition* of $\phi(d_0; p_1,p_0)$ and $\phi(d_1; p_2,p_1)$.

(2) Invertability - If the system is changed from condition 0 to condition 1 and then back to condition 0, the net change in the user descriptors should be zero. Thus,

$d_0 = \phi(\phi(d_0; p_1,p_0), p_0, p_1)$ and
$\phi(.; p_0,p_1)$ may be called the *inverse* of $\phi(.; p_1,p_0)$.

These operators can be derived empirically for various system changes. We hope to go beyond this stage, however, to a point where we can *predict* mathematically the changes that will occur over a wide range of values of system parameters. To accomplish this predictive modelling, we will need to look into some of the mechanisms by which a user actually modifies his behavior, such as

(1) the exchange of one series of commands for another that will accomplish the same goal, but which involves a different mixture of resources demanded;

(2) the exchange of a small number of high demand interactions for a larger number of lower demand interactions which demand the same amounts of resources (the difference being that an error may be found part way through the interactions, making the remainder of the series unnecessary); and

(3) the exchange of user think-time for computer resources (i.e., more careful planning by the user and fewer redundant requests).

Once we have accumulated a sufficient data base of user demands under various conditions, we will attempt to apply optimality considerations in modelling the users' trade-offs. To do this, we shall need to collect sufficient data to map out the possible compensatory interchanges that users can make from various operating points. We shall also need to formalize our notions of the optimality of system operation as discussed in the next section.

4.7 OPTIMALITY CONSIDERATIONS

Using the models discussed above, a manager could investigate the effects of proposed changes in a time-sharing system before committing himself to what might be very substantial capital expenditures. He could compare the improvements that might result from adding more core, from replacing the drum with a faster-access unit, and from other alternatives being considered. If he had well defined measures for judging quantitatively the results of the various alternatives, he could choose the alternative that gave the greatest improvement per dollar expended. In other words, he could optimize system performance within certain financial constraints.

Unfortunately, the optimization of system performance means different things to different people; there are no simple criteria. To the manager of the computation center, optimization involves such factors as

(1) scheduling to achieve maximum utilization of the time-sharing system—e.g., minimizing idle time.

(2) scheduling to maximize the number of users receiving some specified quality of service

(3) scheduling to minimize the delays experienced by a fixed set of users.

To the manager of the staff that uses the services of the computation center, optimization means the maximization of the total job throughput by all users. This is a higher level of optimization than that implied by any of the factors listed above, and is substantially more difficult to treat. Optimization in these terms requires knowledge of the real time behavior of the set of users, not just the computer time spent on various jobs. This level of optimization has received very little consideration in the past. We consider it to be a serious problem; it is by no means clear that optimizing a criterion of concern to the computation center manager will result in the optimization of total real time spent per job. For example, optimizing some internal measure of time-sharing system performance (such as minimizing idle time) is not necessarily equivalent to optimizing the total work throughput of system and users. We offer a simplified, but realistic, example of why this is so.

Consider first a highly idealized time-sharing system that can swap users into and out of core in zero time, and that can carry out all its scheduling activities in zero time. Assume that the users of this system are all identical and, in the absence of other users on the system, would each demand 6 minutes

of CPU time per hour (i.e., 1/10 of the available resources). Then, as the number of such users on the system increases, the observed number of total CPU minutes per hour expended on the ideal system will increase as shown by the dashed line in Figure 9a. For ten or more users the system will be running at 100% capacity. But for more than ten users, the number of cpu minutes per man-hour expended by all users on the system will begin to drop as shown by the dashed line in Figure 9b. This line, of course, is just the dashed line of Figure 9a divided by n, the number of users.

Now consider a more realistic system that spends a non-trivial percentage of time in scheduling, swapping, and core management functions. Such a system might exhibit a CPU minute per hour curve such as the solid line in Figure 9a. For large numbers of users, this system will suffer increasing inefficiencies in scheduling and swapping so that a decrease in CPU time per hour will be observed. Dividing this solid curve by n, we obtain the solid curve for CPU minutes per man-hour shown in Figure 9b.

Before proceeding further, note that in this example the maximum CPU usage per hour occurs with n=14 users. At this point, the system is running at "maximum efficiency" in one sense. But let us look at "efficiency" in a broader sense--one that includes the costs associated with user time, too.

Let us suppose that the users are performing tasks in which useful work is exactly proportional to the CPU time expended, or, more accurately, that each task can be characterized as requiring a fixed amount of CPU time regardless of the real time expended by the user. In reality, of course, it is usually possible for a user to finish a given task using less CPU time if he is willing to invest more of his own time in order to plan his strategy more carefully; let us assume here that this effect is negligible.

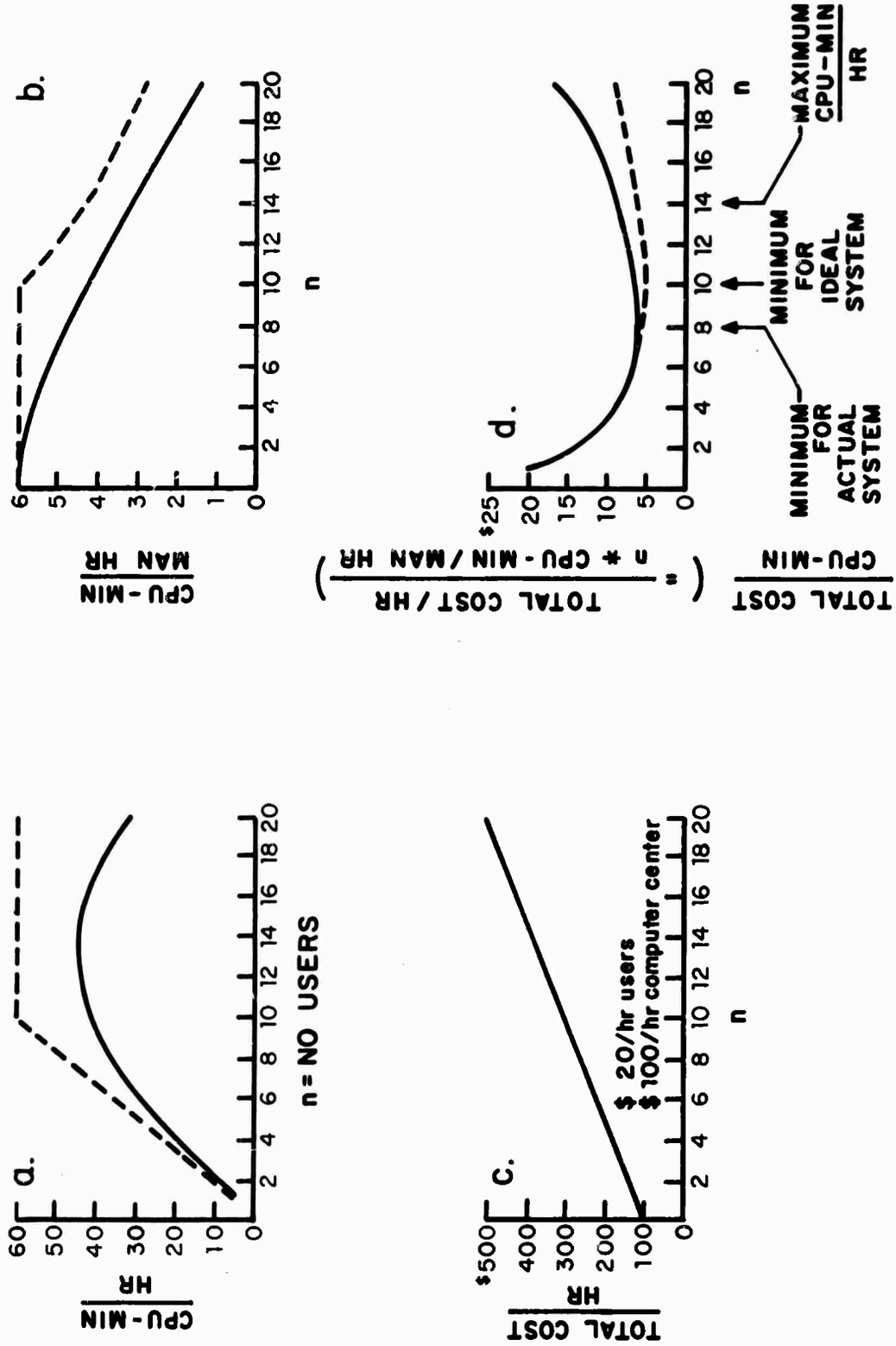


FIGURE 9. EFFICIENCY OF SYSTEM OPERATION UNDER VARIOUS ASSUMED CONDITIONS

Assume that the computer center costs $100 per hour to run regardless of the number of users supported. Assume that users cost $20 per hour in salaries and overhead. Then, the total costs of supporting the center and its users will be as shown in Figure 9c. Now, let us calculate the total cost per unit of useful work performed, i.e., per CPU minute used. This cost is

$$\frac{\text{total cost}}{\text{CPU minute}} = \frac{\text{total cost/hour}}{n * \text{CPU min/man-hour}}$$

Refer to Figure 9d. The dashed line shows the result for the ideal system. Note that the minimum cost per CPU minute occurs for 10 users, the point at which system saturation occurs. For the more realistic system (represented by the solid line), the minimum cost occurs for 8 users and is approximately $6 per CPU minute. Note that the cost of running with 14 users (where total cpu time per hour is maximized) is approxamately $9 per cpu minute, a level 50% higher than the minimum cost!

While these results depend on the numbers chosen and the assumptions made, it appears that for any system exhibiting efficiency characteristics of the form shown in Figure 9a, the minimum total cost per CPU minute must occur at a usage rate below that which maximizes CPU time per hour.

In the course of our experiments, we will be able to measure cpu time per hour for various numbers of artificial users for various combinations of system parameters. We can then explore mechanisms by which a manager might attempt to optimize the overall efficiency of the system and users. We will have to consider such complicating factors as the fact that the number of real users on a system will vary randomly with time of day and with other

factors. It seems to us, however, that this area is an extremely fruitful one in terms of immediate utility of results. We see possibilities of developing improved scheduling strategies to maximize utilization of existing systems and of developing clear-cut procedures for specifying new systems (or modifying old ones) to maximize total efficiency in various applications.

5. EFFECTIVE USER AIDS

5.1 ANNOTATED BIBLIOGRAPHY

Nickerson, Raymond S. and Pew, Richard W. "Oblique Steps towards the Human Factors Engineering of Interactive Computer Systems."

This paper presents a potpourri of human-factors considerations pertaining to the design of general-purpose, interactive computer systems that are meant to be used by nonprogrammers. The reader is warned that it is informal, discursive and opinionated. The intent is to identify some specific problems, to offer tentative solutions to a few of them, and, most importantly, to stimulate more thinking on the part of both system designers and human-factors specialists along these lines.

5.2 REPORT

The paper annotated above is included in this report immediately after this page.

OBLIQUE STEPS TOWARD THE HUMAN-FACTORS ENGINEERING OF INTERACTIVE COMPUTER SYSTEMS*

Raymond S. Nickerson
and
Richard W. Pew

20 July 1971

*Sponsored by the Advanced Research Projects Agency, Department of Defense, under Air Force Office of Scientific Research Contract F44620-71-C-0065, ARPA Order No. 890, Amendment No. 6, Code 1D20.

The authors are grateful to Mario Grignetti for his helpful comments on a draft of this report.

The utility of an on-line, interactive, computational facility that is to be used by nonprogrammers will depend on (1) what capabilities the system provides, and (2) how accessible they are to the user. A scientist, for example, is interested in getting on with his research and is not likely to be enthusiastic about investing much time and effort in acquiring skills that do not have an obvious payoff in terms of his own research goals. There is nothing to be gained by providing him with a sophisticated system that will do many impressive things, none of which he is particularly interested in having done. Nor is there any advantage in giving him a system that will do some of the things he would like it to do, but is prohibitively difficult to use. But what are the characteristics and capabilities that a general purpose, on-line interactive facility should have? And how does one go about implementing them in any particular functional system?

The second of these questions clearly is a technical one, or, more accurately, it spawns a host of problems which must be answered in terms of programming or engineering techniques. The first question, however, is one of human needs and preferences. This being so, it might appear that the answer would be most readily obtained by asking the prospective user what he needs or wants. We think it is not likely to be as simple as that. A realistic appreciation of the features that an interactive system should have is most likely to be obtained as a result of first-hand experience with working systems.

The remarks in this paper are indeed based largely on first-hand experience with a small number of existing interactive systems and a second-hand (reading) acquaintance with

a few others. The treatment of the subject is discursive and informal. No attempt has been made to formalize a set of design criteria or even to map an approach that might be taken to do so. Moreover, we make no claim to exhaustiveness in our enumeration of design considerations. Our intent is simply to identify what appear to us to be *some* of the features that an interactive system should have if it is to be generally useful to individuals whose main areas of interest lie outside the domain of computer technology itself. Many of the design features recommended below are incorporated in one or more existing systems; although, to our knowledge, no single system incorporates them all. Some of the features that will be noted will appear so obviously desirable as to preclude the necessity of even being mentioned. However, that it is painfully easy to overlook what is obvious to hindsight is attested by the fact that operational systems exist in which some of the most clearly desirable features are missing.

It will be evident that we focus primarily on general-purpose, scientifically-oriented—and, in particular, JOSS-like —systems (Baker, 1966). We hope, however, that the reader who is more concerned with special-purpose, problem-oriented, systems—reservation systems, cost-control systems, medical systems, instructional systems—will find some of the discussion germane to his area of interest. The need for effective user-oriented design is especially great in such special-purpose systems, inasmuch as the user is apt to see himself as even further removed from programming and other computer-related activities than is the user of a general-purpose system.

The recommendations that are made constitute a very "mixed bag." They involve various aspects of interactive systems—

languages, facilities, services, dynamics. (We have not paid much attention to the design of user terminals, a topic which is perhaps closer to conventional human engineering than are those which we do discuss. For discussions of some of the human-factors problems encountered in the design of keyboard terminals see Baker, 1967 and Dolotta, 1970. A more comprehensive discussion of human-factors considerations as they pertain to computer input and output devices is contained in Shackel and Shipley, 1970.) We have made no attempt to categorize our recommendations in any way, feeling that to do so would take us beyond the limited objectives of this paper, and perhaps create the impression of a more systematic treatment of the subject than is intended. The recommendations vary greatly in scope and specificity: general design principles are thrown in with "little tricks for making life easier for the user." They are offered quite frankly as opinions, and no effort is made to justify them with experimental data, or otherwise. If they stimulate further thought along these lines, or even the expression of opposing views, they will have served a useful function.

The Cardinal Assumption of the Uninformed User

Efficient interaction with the system should not be dependent on a knowledge of either the internal structure or the details of operation of either the system or the service programs. The user should be free to do his thinking at the level of the language with which he and the computer converse. There should be no need for him to be concerned with the way in which his program is represented within the machine, unless of course it is imperative to him that his program run at maximum efficiency, which usually will not be the case.

Training Requirements and Self-Teaching Capabilities

The system should require very little off-line training or instruction of the user. Ideally, it should be designed so that a novice can use it, at least haltingly, after spending a few minutes with a tutor or a manual, and can expect to learn to use it efficiently from the feedback provided by the system itself. Insofar as possible, the system should be designed in such a way that the most efficient and powerful approaches to problems are readily discovered by the user in the process of interacting with it. That is to say, the system should have a built-in teaching capability designed to facilitate the acquisition of that knowledge and those skills that qualify a user as an expert.

For example, it would be helpful to the novice user to be able to request the computer to give him examples of types of statements whose format he has forgotten, or not yet learned. To illustrate: a beginner might realize that the language allows "if" statements, but may not be able to put into an appropriate format a particular conditional that he wishes to write. He would then like to be able to put the system into a "teach" mode and ask it to give him some illustrative "if" statements—perhaps by simply typing "TEACH IF." The computer could thereupon produce a sequence of "if" statements in an order of increasing complexity until it had either satisfied the user or exhausted its supply of examples. Such a feature would also serve the more experienced user, who from time to time needs to refresh his memory regarding allowable statement formats.

A common practice is to build format information into the error diagnostics. For example, a format error might elicit a

remark from the computer such as "The correct format is:" followed by an example of a correctly formatted statement representative of the type that the diagnostic program thinks the user was attempting to write. The objection to this procedure is that, if an experienced user is at the console, the lengthy output may be not only unnecessary but even bothersome. He may know exactly what his error is the moment it is pointed out to him that an error has been made. It would be in keeping with the policy of eliminating noninformative computer-to-user messages (see below) to provide the user with illustrative statements and detailed error diagnostics only in response to an explicit request.

Prompting can be another useful teaching technique and memory aid. To log in to the TENEX system,* for example, the user must type, in order and with appropriate terminators, the word "LOGIN," his name, a "password" and a job number (the latter for billing purposes). The experienced user does this more or less automatically; however, the novice or infrequent user can easily violate the format requirements, enter items in the wrong order, or forget to enter an item altogether. TENEX facilitates entry by identifying each of the components of the log-in procedure (except the first). The user need remember simply to type "LOGIN," followed by a special terminating symbol (the "escape" key on the teletype in this case). The computer will

*TENEX is a time-sharing system implemented on a DEC PDP-10 computer at Bolt Beranek and Newman Inc. Several of our examples are drawn from this system, in part because we happen to be familiar with it and in part because considerable attention was given to human factors problems by its designers. For descriptions of the system, see Myer and Barnaby (1971) and Burchfiel and Leavitt (1971).

then type "(USER)" and wait for the user to type his name, whereupon it will type "(PASSWORD)", and so on. The experienced user can suppress this prompting simply by using a different terminating symbol.

Updating Information

The need to train the neophyte is one requirement that occurs to everyone. A less obvious training requirement concerns the continuing education of the experienced but sporadic user. Few interactive systems are static. New procedures and upgraded versions of old procedures appear regularly. The chronic user who is on the system much of the time will assimilate changes gradually as they occur. The infrequent user will find it much more difficult to accommodate to changes that have occurred during a period of a few weeks or months that he has not used the system.

Typically this kind of training is provided by announcements made at sign-on time for two or three days following a change, and a memo to users may be issued to be read at their convenience. A better procedure would be to provide communication about system modifications contingent on their need. If a new format or command is defined that replaces an old one, the user should be trapped to a brief description of the new one and how to use it whenever he attempts to execute the old one. This procedure is rather like that used to correct for the dialing of an out-of-date phone number: the operator interrupts and provides the new number. When new procedures are introduced that supplement rather than replace others, use of the basic command should call forth a description of the supplemental procedure prior to execution of the command for the first three or four times the user

applies it. The important point is that the critical dimension relating to the need for prompting the user's memory is not the time since the system change was made but the number of times that particular user has already been reminded of that change, and perhaps the recency of the last reminder. Such a procedure implies a bookkeeping burden for the executive program, but one that could be easily managed in a good system.

One simple expedient for getting updating information to users who need it, without forcing it on those who do not, would be to have the computer type the date (or perhaps the number) of the last change in the system, whenever anyone logs in. If the user is already aware of the change, he will simply proceed with the work session; if not, he can ask for a report. Following the typing of the report the computer would then give the date of the next-to-last change, and again, the user can decide whether he needs, or wants, to know about it. And so on.

Computer-to-User Messages

Computer-to-user messages should be designed to accommodate users representing all degrees of familiarity with the system. There are two types of computer-to-user messages that may occur in an interactive session: (a) those which the user intentionally elicits, either by requesting some specific outputs (program listings, values of variables, etc), or by inserting messages of his own composition into the body of his program, and (b) those that are preprogrammed into the basic system. We shall be concerned here only with the latter.

The purpose of such messages is to convey to the user some information that will facilitate his further progress with his

program. Most commonly, they take the form of requests for specific inputs, of information concerning the state of the system, or of error diagnostics. In the latter case, an indication that an error has been made may or may not be accompanied by some information concerning the probable nature of the error. The problem is that of designing a message set and rules for message generation that satisfy the needs of users who represent every possible level of expertness in their interaction with the system. Novices will require lengthy messages which are completely self-explanatory; experts will prefer coded outputs which are as brief as they can possibly be made. Ideally, for the novice, every message should be meaningful the first time it is encountered. Satisfying this desideratum is in keeping with the objective of minimizing the amount of training a beginner must have before interacting directly with the system. It means, however, that messages should be written in a natural language (e.g., English) in whatever detail and with whatever degree of redundancy are necessary to ensure that they will be readily understood. Detail and redundancies that are helpful to a user who is learning the system will become sources of irritation, however, as he acquires skill. (One of the most reliable marks of the experienced user of an on-line system is his tendency to be exasperated by any delays which he perceives to be unnecessary. Given the opportunity, he would invariably replace lengthy messages with the briefest possible codes.) Even for experienced users, however, it is imperative that the computer do *something* whenever it receives a command that it cannot interpret. This is essential if one is to avoid the situation in which the computer is waiting for the user to input something interpretable, while the user is waiting for the computer to operate on what he assumes was an interpretable input.

Several possibilities suggest themselves for coping with the problem of conflicting desiderata of novices and experts concerning the form and content of computer-to-user messages.

1. Two separate programs. One possibility is to keep on hand two entirely independent systems which differ primarily, or only, with respect to the computer-to-user messages they generate. In one case, the messages, being complete and, hopefully, self-explanatory, are designed for the novice, the occasional user, and the visiting observer. In the other case, the messages are greatly abbreviated and intelligible only to the programmer or the user who has had considerable experience with the system.

2. One program, two message sets. It is, of course, oversimplifying things considerably to recognize only two types of users: novices and experts. It is more realistic to recognize that users represent a full spectrum of expertness. Any particular user masters a system only slowly over a long period of time. Moreover, different users, because of their own particular needs, may acquire skill with some aspects of a system while remaining relatively unskilled with respect to others. It may be advantageous, then, to allow the user himself to decide when he wishes to be treated as a novice, and when he wishes to attempt to play the expert. A simple way to provide this option is to include two complete message sets in the system, and to allow the user to switch at will between one and the other. Presumably, given such an option, the amount of time the user spends in the novice mode will decrease fairly regularly as he gains experience with the system.

3. "Yeah, yeah" signal. A third possibility is to provide the user with the means of cutting short a computer-to-user message while it is being typed out. For this approach to be effective, the user should be able to terminate any message, by pressing a single key, at any time during the message typeout. With this capability, the user need attend to the typeout only so long as it is informative. How much of a message he will want to see will depend, of course, on his familiarity with the system. Presumably, one's use of the interrupt option will become more frequent and more rapid as his experience with the system increases.

4. Two-part messages. A fourth possible approach is to (a) store each computer-to-user message in two forms—a concise mnemonic code and a complete self-explanatory statement, (b) *always* output the coded form of the message first, and (c) output the self-explanatory statement only if the user requests it, say, by responding to the coded form with "?". The advantages of this approach are several. First, the same program and the same mode of operation are appropriate for all users. Second, although decoded messages are always available when desired, the user never receives a lengthy message unless he specifically requests it. Third, the procedure facilitates the acquisition of just that knowledge which will make time-consuming messages unnecessary.

A combination of (4) and (3) would provide a particularly accommodating facility.

String Recognition

The capability for the computer to perform recognition on a partially complete character string effectively combines the principles of concise computer-to-user messages, prompting, and efficient training procedures. The string recognition procedure that is implemented in the TENEX system works in the following way. Whenever the user thinks that he has typed enough of a command string or file designator so that the intended command or file is uniquely specified, he may terminate the partially completed string with one of several terminators. With one terminator the computer either completes the typing of the designated string and waits for the next entry or parameter, or, if it cannot identify uniquely the string that has been terminated prematurely, it rings the terminal bell and awaits further input to complete the string. In a second termination mode the system accepts the abbreviation as it stands and either executes the command directly, or, if it cannot recognize the command or make a unique selection, it prints a "?" and aborts. In an earlier version of this recognition feature the computer took over for the user as soon as it had received sufficient characters and completed the string automatically. Given this procedure the user finds it easy to type accidently more than the requisite number of characters before the computer has time to take control. The result may be the typing of a few stray characters at the end of the command that at best are misleading and at worst confound the beginning of the next input. The string-recognition feature, as currently implemented in TENEX, is especially convenient if it can be applied to terms defined by the user himself as well as to system-defined commands.

Default Values and Conditions

Often in interperson conversations, information is exchanged by default. If one mentions Paris, for example, it is likely to be assumed that he is referring to Paris, France; had he meant Paris, Maine, he would have been expected to say Paris, *Maine*. Similarly, in the case of man-computer interaction it is sometimes possible to assume what unstated values of program parameters should be, and to assign them by default whenever the user does not explicitly indicate otherwise. Default conditions make it possible to build into the system considerable sophistication that can be exploited by the user as far as he wishes, or to the degree consistent with his level of training. As an example consider the file designation procedure used by the TENEX system. A complete file designator consists of five parts, and might look as follows:

ALPHA. F4; 3; A12345; P7752Ø2

Part I (ALPHA in our example) is the file name assigned by the user. The system will recognize an abbreviation (first few letters) of the name so long as no other file name would be abbreviated the same way. Part II (F4) is the file extension, which tells the system what kind of file is involved. It is also subject to the automatic recognition procedure. Part III (3) is the version number. When creating a new file the default value of the version number is one. When creating a new version of an old file the default value is one greater than the last number used with that file name and extension. When deleting a file the earliest version number is assumed unless the user explicitly specifies a higher one. Part IV (A12345) is the account number to which page charges will be assigned. If the

user defaults this number, the account to which his compute time is charged is assumed. Part V (P7752Ø2) describes a protection or privacy status for the file. If no number is specified it is assumed that any other user may read the file but only the creator of the file may write into it or delete it. Note that for a typical user Parts I, II and occasionally Part III are sufficient to declare most files and it is the exception that requires further specification.

In some cases in which it is not clear in advance what the best default value is, it might be appropriate to sample user opinion or to collect statistics on the most frequently used value in order to determine what it should be. When it is important for the user to know exactly what he defaulted, the machine should prompt him with the defaulted value. It is important, for example, for the TENEX user to know his extension and version number, but the account and protection information are not displayed unless specifically requested.

Program Component Identification

There should be a straightforward way of structuring a program and of identifying its components. Perhaps the most common structure in conventional programming is that of a heirarchy: programs, subprograms, routines, subroutines, etc. There is every reason to expect that this will be equally true of interactive programming; hence, there is need for a means of identifying program components in such a way as to make it possible to refer to any level in a hierarchy of arbitrary depth.

Several of the current JOSS-like systems provide for a two-level organization of a program in "parts" and "steps." The convention is to identify steps with decimal numbers, the integer part of the number designating the part to which the step belongs. Reference can then be made to, and operations performed upon, either individual steps or parts as wholes. Thus, for example, the command "DELETE PART 3" would, in effect, delete steps 3.1, 3.12, 3.2 and any other steps identified with a number whose integer part is 3. The restriction of two levels imposed by this scheme might not be a serious limitation for the casual user of a system; however, it probably does represent an unnecessary constraint for the more experienced user. Moreover, it is a limitation that is removed by simply making the convention that when a command can appropriately reference more than a single step (e.g., DELETE, TYPE, DO), the command will be understood to refer to all steps whose most significant digits correspond to the number in the command statement. Hence, the command "TYPE PART .1324" would cause the typing of steps .13241, .13242, .132431, and any other step whose number began with .1324. If the user wished to refer to a single step, he would, of course, have to use enough digits to identify that step uniquely. For example, assuming that his program contained each of the above step numbers, in order to have the single step .1324 typed, he would have to say "TYPE .1324Ø."

List-processing languages such as IPL and LISP are not organized in terms of numbered steps, so this convention does not apply. In LISP, program components are "symbolic expressions," each of which is comprised of a function and its arguments. The arguments of a function may be functions in turn, so that these programs also have a hierarchical structure. Expressions

or subexpressions may be identified via the appropriate function names. List-processing languages are less likely to be of concern to the nonprogrammer computer user than are the JOSS-like languages—at least in the near future—so they are given little attention here.

Editing Capabilities

The system should provide flexible editing and error-correcting capabilities. It is convenient to make a distinction between two broad classes of editing and error-correcting operations: those which may be perfcrmed on a program component or step as it is being composed, or *local* operations, and those which may be performed on steps which have already been inserted into the program, or *remote* operations.

There are two local operations which, from the user's point of view, are needed: one to delete the last character typed, and one to delete the entire step or program component currently being entered. Each of these should be executed by striking a single-control character. The operation deleting the last character should be iterative, allowing the user to delete the last n characters typed. In the case of teletype or typewriter input it should not be possible, with this operation, to delete elements past the first character of the current line or program component because it becomes very difficult to keep track of exactly what was deleted. This restriction is not important in the case of a CRT terminal where the consequences of deletion can be portrayed literally to the user; i.e., the deleted characters actually can be made to disappear and new ones to appear in their places.

When text is being displayed on a CRT as it is being typed, a cursor or underscore should be used to show the location of the next character to be typed. This is especially helpful when nonprinting characters (spaces, tabs, carriage returns) are being used in formatting tables, labeling graph axes, etc.). A further convenience to the user would be an alternate mode of display in which nonprinting characters are explicitly represented by special symbols.

A flashing cursor can be helpful when backspacing over displayed characters for erasure or editing. Rule: have the cursor flash whenever it is pointing to the location of a character that has just been deleted from memory. Again this would be particularly useful in the case of nonprinting characters.

There are four remote editing operations that are essential to an on-line system. They are the operations of deletion, replacement, insertion, and revision. The operand may be a variable, a step or other program component. Given a step-numbering scheme such as that described above, the remote operations of step deletion and insertion are self-evident. One advantage of such a scheme is that it obviates the renumbering following the deletion or addition of steps. For example, given a program comprised of steps .11, .12, .13, and .14, deletion of step .12 and insertion of two additional steps between .13 and .14 would not necessitate renumbering any of the original steps that are retained, even though their ordinal positions in the program have been changed. The steps of the program following the indicated changes might be numbered .11, .13, .131, .132, and .14. Step replacement would be accomplished by simply writing a new step and assigning it an old number, the system being designed

so that whenever a step is given the same number as that of a previously entered step, the original step is replaced by the new one.

The delete operation can of course cause grief when supplied with an erroneous argument. An easy way to guard against this event is to force the user to think twice about any such command. In PROPHET (Castleman, *et al.*, 1970), a CRT-oriented chemical/biological information-handling system, the effect of a delete command is to have the to-be-deleted element blink on the display. The user then must verify that the blinking element is in fact the one that he wishes to delete.

A system that allows only the three remote operations of deletion, replacement, and insertion would be reasonably adequate for many applications; however, to be truly efficient, it should include, in addition, a capability for revising steps or other program components without completely retyping them. In many instances the user will want to change only those portions of a step that are in error, while retaining those portions that are correct. It is an inconvenience, for example, to have to retype a lengthy and involved algebraic statement to correct a single erroneous character. The need here is for deletion, replacement, and insertion operations which can be performed on *elements* within a step. The more sophisticated systems provide editing commands for searching program components for particular characters or character strings, and for performing delete, replace, or insert operations relative to the result of the search.

In addition to providing these component editing capabilities it is also important not to place artificial constraints on the

ways in which they may be used. It should be permissible to intermix freely editing commands and to make up strings of commands to be executed as a unit. For example, to change N=N+1 to N=N+2, one might want to write an editing procedure that would search for the string N=N+, delete the next character in the line and insert 2 in its place. In the TENEX version of TECO, which is a language used primarily for the purpose of editing, this is accomplished by typing the string

SN=N+$DI2$$

where the S, D and I indicate search, delete and insert, respectively. The first and second dollar signs terminate the search and insertion strings, and the third executes the string of editing instructions.

A common practice in algebraic interactive languages is to reject an input string if the computer detects a syntactic error and to inform the user of why the input was unacceptable. We recommend instead that the aberrant string be retained in the buffer and the computer automatically shifted into an editing mode so that the user may choose to delete the entire string or, if possible, to correct it by changing one or two erroneous characters. It is more than mildly irritating to complete the typing of a complex algebraic expression only to find that it must be completely reentered in order to add one forgotten right parenthesis.

Direct and Indirect Commands

The system should allow both direct and indirect commands.

By direct command is meant a command that is to be executed immediately; an indirect command is one that is to comprise a component of a program, and that will be executed in the course of the execution of the program to which it belongs. The direct-command capability allows the computer to be used as a powerful desk calculator for such purposes as evaluating mathematical expressions, generating tables, and plotting functions on a one-shot basis. It also serves as an important tool for debugging and editing active programs. Indirect commands provide for the construction of programs. Virtually all conversational languages include both direct and indirect commands. In some cases, however, direct commands comprise a minimum set (DO, RUN, EXECUTE), in which case in order to use the computer as a desk calculator one must enter an indirect command and then execute it as a program.

Arbitrary Starting Point

The user should be able to start or restart his program at any point. In particular, after fixing an error that has caused a running program to halt, he should be able to restart the program at the point at which it stopped.

Variable Names

In composing programs, the user should be free to assign names to variables in a way most consistent with his own mnemonic conventions. Ideally, he should be allowed to call variables anything he wants; in practice, other considerations may place a limit on the number or types of characters a name may be allowed to contain. If a limit must be imposed, five or six

characters per name would probably be adequate for most users; three characters per name is perhaps tolerable; a single character limitation (even with subscripting) is a definite handicap.

Language Modification and Abbreviations

A means should be provided for the user to modify the language and redefine terms. For example, an individual who finds himself using a small set of commands very frequently might find it economical to replace each of these commands with a single-character abbreviation. Insofar as possible, he should be allowed to establish equivalences of this sort.

One should also be able to define and use abbreviations for such things as variable names. For example, PROPHET, the chemical/biological information-handling program mentioned above, permits one to give a variable such a name as "MOLECULAR FORMULA OF ASPIRN," and then define and use an abbreviation such as "MA" (Castleman, *et al.*, 1970).

The user should not, of course, be allowed to make language changes that will affect other users in any way.

Address Arithmetic

Languages for which a step is the basic program component (e.g., JOSS-like languages) should permit the changing of step numbers for any specified program segment with a single command. For example, a command like "CHANGE STEPS .21 to .46" might be used to replace all the step numbers beginning with .21 to new numbers beginning with .46, leaving the less significant digits unchanged.

Algebraic Expressions as Inputs

The system should accept and correctly interpret any evaluatable algebraic expression in any case in which a number is an admissible input. As a simple but important example, one should be able to input fractions as *fractions*, that is, one should be able to insert 1/17 as opposed to .Ø58889. The importance of this capability does not stem from the fact that a fraction is easier to type than a decimal (although if one wants accuracy, he will, in general, have to type several more characters in the latter case), but rather from the fact that, if the user has the fraction to begin with, converting it to a decimal number involves a task that the computer, not he, should perform. The ability to input fractions directly is a particular advantage to the user who is dealing extensively with probabilities.

Identification of Precision Limits

The limitations of the system with respect to numerical precision should be explicit in the output. The system should not produce numbers with more significant digits than are justified by the computational accuracy of its number-handling procedures. For example, if the system can assure only ten bits of accuracy in its number representation, it should not output numbers with more than three significant (decimal) digits. Since most machines use floating-point arithmetic, which allows the manipulation of numbers whose magnitude is far beyond the precisional limits of the system, there must be some straightforward way to represent arbitrarily large numbers so that the accuracy limitation is obvious. One possibility is to express

all numbers in scientific notation with the fractional part being limited to the number of digits implied by the precisional capabilities of the system. Another possibility is the use of filler symbols. For example, given a limitation of three decimal digits of accuracy, the number 365,741 might be represented as 366,xxx. It should not be represented as 366,ØØØ, since in this case the limitation is not obvious. The system should round the output to the least significant digit; it should not truncate. In short, when a user receives a number from the computer, he should be able to assume that it is exactly the number that he would have obtained had the computation been done by hand, and rounded off to the same number of significant digits.

Formatting Options

The system should provide formatting options specifically designed to assist the user in making his program easy to read. Extra spaces and carriage returns should be freely allowed and should be preserved in storage at the level of the symbolic program. In scientific programming, one frequently wishes to construct algebraic statements involving several depths of nested parentheses. Parenthesizing errors are very easy to make, and can be frustratingly difficult to find. It would be a help to have several, say three, different characters, e.g., (, [, {, for formatting algebraic statements. These characters could be equivalent as far as the program interpreter is concerned, but the distinction should be maintained at the level of the conversational program. Such a feature would facilitate the construction of complex algebraic statements and would simplify the process of finding errors when they occur. It would be particularly helpful if the different parenthesizing symbols were different sizes.

Another useful formatting convention, easily implemented with a typewriter as the I/O device is that of color-coding the dialogue, printing user-generated text in one color and computer-generated text in another (Baker, 1966).

Procedure Definition

There should be a straightforward means of defining and storing generalized program components and retrieving them for incorporation as elements in programs or higher-order components. Having once written a particular generalized program component (procedure, function, macro, subroutine), one should not have to write the same component again. Heavy users of an interactive system are likely to be developing many programs having common components. The prospect of developing a library of program components especially tailored to one's own needs is perhaps one of the most compelling enticements that a computer system can offer to a prospective user.

Procedure Library

The system should maintain a central public library of programs and procedures that are available to all users. The library should be designed to expand as users generate new programs of general interest. Every user should have read-only access to the library on a continuous basis. He should not, however, be able to enter programs directly into the library. One possible scheme for allowing a user to contribute to the library would be to have him deliver a program to a temporary file which is periodically examined by the system supervisor or librarian for the purpose of updating the library file.

Compilation Capability

A system designed specifically for scientific and engineering applications probably should have a compilation capability. The interpreter should be used for exploratory programming; however, when a program is to be used frequently for production runs it should be compiled. This is especially true when compilation results in noticeably shorter system response times. It is essential, however, that such a compiler accept as input the program as it was written for the interpreter.

File Storage

In cases where lengthy work sessions are anticipated, it should be possible for the user, when terminating a session with work unfinished, to leave the system in such a state that, upon reentering it at a later time, he will be able to resume his work exactly where he left off. This means providing the user with the capability to store his virtual core in a long-term storage medium such as magnetic tape or disc, and to retrieve it upon reentering the system. The user should also be able to maintain files of his own subroutines, programs and data sets.

Short Interruptions

In addition to the capability for the resumption of work after indefinite periods, there should be a simple procedure for allowing brief interruptions in a work session. It frequently happens in the course of an on-line session that the user finds it necessary or advantageous to leave the console temporarily (e.g., to attend to an unexpected visitor or telephone call, or

to dispose of some pressing business—or perhaps to cogitate about his program or some results he has obtained from running it). If it is likely to be several minutes before he will return to the computer, and particularly if he is being charged on the basis of on-line time, he will want in such cases to be able to take "time out," to tell the computer it can forget about him until such time that he indicates that he is ready to resume the session. The procedure for effecting such a recess should be less involved than that used to store a system for reactivation in the indefinite future. It should not, for example, be necessary explicitly to create files on a long-term storage device. Ideally, to initiate the time out, the user should be required to do nothing more complicated than to press a special function key, or perhaps to type "time out" or "wait" or some such thing. Resumption of the session should be effected by an equally simple procedure.

Program and File Information

The system, on request, should be able to provide the user with information concerning the status or contents of his program. It should be able to produce, at the minimum, a copy of any specified segment of the user's program, a list of variables, functions, procedures, macros that the user has defined, a table of contents of the user's files or previously stored programs, values of variables, indexes, subscripts, etc.

Status and Control Information

The user should be provided continuously with status and control information. At the very least, he should be informed

as to whether he is waiting for the machine or it is waiting for him. (The JOSS system provides this information via a red and a green light at the console that indicate whether the computer or the user is controlling the typewriter [Baker, 1966].) Given that the user is waiting for the computer, he might like to know: (1) is the computer currently working on his problem? (2) is it waiting for a peripheral device like a tape unit or line printer? (3) is it waiting in a queue for its "slice" of time? or (4) is the system dead?

Feedback to the user is particularly important when the length of the delay to be expected is unknown. For example, a long pause after some data have been entered can make the user wonder if he has entered data incorrectly, or possibly has not properly signaled the computer that he is done. The computer should signal receipt (or acceptance) of entry immediately, even though there may be a delay before the next entry can be accepted, or before there is a substantive response (Poole, 1966).

In some systems it is practical to include an auxiliary display at the terminal that provides the user with his current status with respect to these alternatives, but in systems operating over telephone lines this may not be economically practical. An alternative that seems to be quite effective is to provide a status command with which the user can interrupt the ongoing computation long enough to have printed a computer-to-user message describing both his current status (running, I/O wait, etc.) and give the cumulative log-on and CPU time used to date. The system is then restored immediately to its former status with no loss of priority. In the course of a long computation, user-initiated periodic status interrupts of this sort can provide quantitative information regarding how much of the machine's time one is getting per unit of elapsed time.

The system should be able to tell the user how much time he has used since the beginning of the session, or since some specified date. It should also be able to produce a statement of charges accrued since the beginning of the current billing period against the user's job number or account.

System Dynamics Information

If the system dynamics (e.g., response time) change significantly with the load, as they usually do, it would be a convenience to the user if he could get an indication of what the load is before deciding whether he should get on. At a minimum the system should be able to answer the question: How many users are now on line? Other, and more helpful, items of information are, in principle, obtainable (e.g., mean system response time to a request for a given time slice over the last n minutes), but only at a somewhat greater cost in overhead program execution.

Fail-Safe Provisions against Potentially Fatal Operations

Users make mistakes. They enter commands they did not intend and sometimes discover what they have done too late to avoid the dire consequences. If one deletes a program, or a file, by mistake, for example, in most systems there is no provision for recovering from such an error. The program, or file, is gone and would have to be reentered in its entirety. Provisions can be made, however, either for decreasing the probability of such errors or for facilitating recovery from them when they do occur.

A simple measure for decreasing the probability of such errors is to require for commands that modify stored programs or files (e.g., DELETE, KILL, MODIFY) some confirmation in

addition to the usual command terminator. This is tantamount to forcing the user, after issuing a potentially destructive command, to indicate explicitly, "Yes, that is what I meant to say." Such a fail-safe measure is implemented in the PROPHET system, as noted under Editing Capabilities, above.

An alternative way of dealing with this problem is to implement procedures for recovering from the erroneous entry of potentially disastrous commands. Some systems, for example, have implemented "UNDELETE," "UNDO," or "RESTORE" commands. In BBN TENEX, UNDELETE restores the file to the user's directory after it has been inappropriately deleted. It may be used any time up until the user logs out of the system, at which time his deleted files are expunged. In TENEX-LISP, UNDO undoes the effects of a program execution. "UNDO PART 4" would restore the program to the status it had, complete with the variables and constants as they were, before part 4 was executed.

No Invisible Mistakes

Interactive systems make frequent use of nonprinting characters as control characters. It is important to the user who is attempting to diagnose an error that it not be possible to have an error hidden because it involves the application of a nonprinting character. This can be avoided by having a special character echoed at the terminal for every one that does occur in a character string.

Conditional Dump of Stacked Input

When a full duplex terminal is in use, in which the type bar is controlled by the computer and every typed character is echoed

through the machine, it is possible to type at the keyboard while the computer is occupied with computation. The typescript that is entered this way is not reflected back to the terminal until the computer releases control of the interaction. If the computation is ended appropriately all is well, but if the computation is terminated prematurely because of an error or because of an unanticipated program branch, then the preentered typescript is appended to the end of the error message and is interpreted as the beginning of a new, but, in this case, inappropriate message. Whenever an error termination like this occurs, the system should automatically dump the prestored typescript and leave the user with a clean slate to deal with the error condition.

Report Quality Output

The system should be capable of producing output of a quality acceptable for incorporation in official reports. This goal is somewhat more easily realized with typewriters or with MODEL 37 teletypewriters than with MODEL 33 or 35 teletypewriters, since in the former cases one has a conventional character set, including both upper- and lower-case characters. There is, however, a considerable need for research into the problem of improving the design of keyboard devices that are to be used as computer terminals (see Dolotta, 1970). The identification of an adequate character set is only one of the many problems that arise in this context.

"Sense" Switches

Most computers provide the programmer with a set of toggle switches (usually referred to as "sense switches") on the console,

each of whose positions (up or down) can be examined by the program. By making the course of the program at different points contingent on their positions, the programmer can make it possible to control the flow of his program at run time by manipulating the appropriate switches. Such real-time control of a running program could be a very great convenience to the user of an interactive system, and could be provided by means of a set of sense switches located at the remote terminal. A cutout overlay that accompanies the program to be run could be used to remind the user of the status and meaning of each sense switch, which could change, of course, as a function of the program being run.

User Interrupt

We may think of the user-computer interaction as always being under the control of either the user or the computer. Whenever it is the user's turn to "say" something, we say he is in control. He may actually be typing a user-to-computer message, or he may be scratching his head thinking about what to type; in either case, if the computer is waiting for an input from him, we say he is in control of the interaction. Similarly, the computer, while in control, may be outputting a computer-to-user message, or it may be executing a program in preparation for outputting a message. Normally, control passes either from the user to the computer, or vice versa, at the termination of a message. That is, one of the communicants regains control by virtue of the fact that the other relinquishes it, having completed a message, and having nothing more to say at the moment. To a large extent, it is this continual exchanging of control, the give-and-take dynamics of the situation, that justifies describing the interaction as "conversational." There

is a need for one exception, however, to the normal way of passing control from the computer to the user: the user should have the ability to interrupt. That is, he should be able to seize control of the interaction at any time, without waiting for the computer to relinquish it.

The need for this capability is most clearly seen in the case of a lengthy computer output which, from its beginning, is obviously erroneous. Suppose, for example, that the user has programmed a loop to generate a lengthy table, and that by the time the first few values of the table have been typed, it is clear that there is something wrong with the algorithm. In such a case, the user should not be forced to wait until the entire table has been generated before regaining control of the interaction. He should be able, by pressing a single key, to cause the computer to stop what it is doing and to await further instructions from him.

Background Execution Option

The efficiency of an interactive system could be increased by providing the user with the option of "detaching" his program from interactive control at the terminal and having it run as a low-priority background process. Suppose, for example, a particular application involves developing a procedure for generating fairly lengthy tables. While developing and debugging the procedure, the user wants to be on-line. Once the procedure is operating satisfactorily, however, he may simply want to leave it alone and let it generate its output. In such a case, the user would like to be able to leave the terminal and return after the tables have been completed. Moreover, unless there is some urgency for an immediate result, he would probably be

content to have it generated at the computer's leisure, especially if background-processing time were charged out at a lower rate than on-line time.

Programmed Logout

There should be an instruction to discontinue service that could be appended at the end of a program, thus permitting the user to log out of the system and disconnect the terminal indirectly. If one has written a program that will run for a considerable time without intervention, it should not be necessary for the user to stay around simply to pull the plug at the end of the session. As a fail-safe protective measure against program malfunction, it would be a convenience for the user to be able to specify a time at which his program should be automatically terminated in the event that it is still running.

Complaints and Suggestions

The system should have a complaint or suggestion input capability. Ideas for system improvement frequently occur to a user in the process of interacting with the system, and are forgotten by the end of the session. Similarly, a minor malfunction, unless it is serious enough to terminate the session, is apt not to be remembered. It would be a convenience to the user, and it should be an aid to the system managers, if it were possible to insert a complaint or suggestion directly into an appropriately designated file at the point during the on-line session when the occasion arises. A hard-copy record of the file could then be made periodically and might prove to be a valuable source of information when attempting to improve the system.

REFERENCES

Baker, C. L., 1966. JOSS: Introduction to a helpful assistant. *Memorandum RM-5058-PR*. The Rand Corp., Santa Monica, Calif.

Baker, C. L., 1967. JOSS: Console design. *Memorandum RM-5218-PR*. The Rand Corp., Santa Monica, Calif.

Burchfiel, J. D. and E. M. Leavitt, 1971. TENEX: User's Guide. Bolt Beranek and Newman Inc., Cambridge, Mass.

Castleman, P. A. *et al.*, 1970. THE PROPHET SYSTEM: A final report on Phase I of the design effort for the chemical/biological information-handling program. National Institutes of Health and Bolt Beranek and Newman Inc.

Dolotta, T. A., 1970. Functional specifications for typewriter-like time-sharing terminals. *Computing Surveys*, 2, 5-31.

Myer, T. H. and J. R. Barnaby., 1971. TENEX: Executive language manual for users. Bolt Beranek and Newman Inc., Cambridge, Mass.

Poole, H. H. *Fundamentals of Display Systems*. Spartan Books, MacMillan & Co., 1966.

Shackel, B. and P. Shipley. Man-computer interaction: A review of ergonomics literature and related research. EMI Electronics Ltd., Report No. DMP 3472, Feb. 1970.