Proc. Spring Joint Computer Conference, 1964

A GENERAL-PURPOSE TIME-SHARING SYSTEM

Jules I. Schwartz, Edward G. Coffman, and Clark Weissman
System Development Corporation
Santa Monica, California

INTRODUCTION

Since June 1963, a Time-Sharing System has been operational at the System Development Corporation in Santa Monica. This system was produced under the sponsorship of ARPA and has utilized ideas developed at both Massachusetts Institute of Technology[3,4] and Bolt, Beranek, and Newman,[1,11] as well as some original techniques. Time-sharing, in this case, means the *simultaneous* access to a computer by a large number of independent (and/or related) users and programs. The system is also "general purpose," since there is essentially no restriction on the kind of program that it can accommodate. The system has been used for compiling and debugging programs, conducting research, performing calculations, conducting games, and executing on-line programs using both algebraic and list-processing languages.

This paper is divided into four major discussions. These are: (1) an outline of the capabilities provided for the user by the equipment and program system; (2) a description of the system's operation, with an analysis of the system scheduling techniques and properties; (3) a somewhat detailed description of two of the currently operating system service programs; and (4) a conclusion and summary.

CAPABILITIES FOR THE USER

Equipment Configuration

The major computer used by the Time-Sharing System (TSS) Executive is the AN/FSQ-32 (manufactured by IBM). Also used by the system is the PDP-1 (manufactured by Digital Equipment Corp.), which is the major input/output vehicle for the various remote devices.

The remote input/output devices available to users include Teletypes, displays, and other computers. These devices can be run from within SDC, and from the outside, with the exception of displays, which can be operated only a short distance from the computer. It is expected that computers to be used at remote stations will eventually include the CDC 160A, the DEC PDP-1, and the IBM 1410. (Currently only the 160A is being used, from an installation 400 miles distant from the Q-32.) Figure 1 is a description of the system's remote equipment configuration.[1]

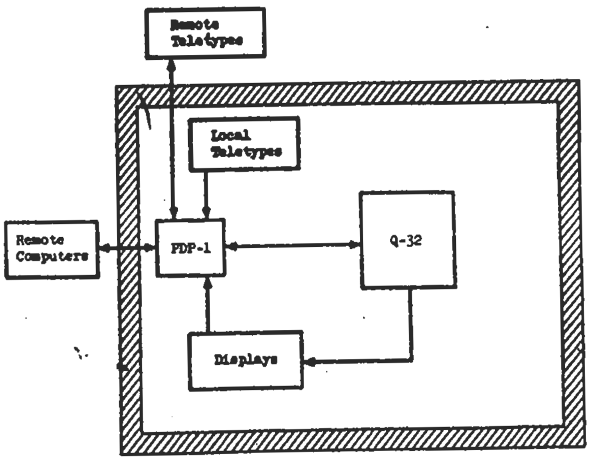

Figure 1. Remote Equipment Configuration.

The AN/FSQ–32 computer is a 1's-complement, 48-bit-word computer, with 65,536 words of high-speed (2.5 usec. cycle time minus overlap) memory available for programs, and an additional 16,384 words of high-speed memory available for data and input/output buffering; the latter memory is called input memory. The PDP-1 also has access to the input memory; thus, this memory serves as the interface between the two computers. In addition, the Q-32 has an extremely powerful instruction repertoire, including access to parts of words for loading, storing, and arithmetic; it also has an extensive interrupt system.

Figure 2 shows the principal components of the system and the important information-flow paths throughout the system. As implied in the figure, each main memory bank (16K words) is individually and independently accessible by three control units: the central processor unit, the high-speed control unit, and the low-speed control unit. High-speed I/O, low-speed I/O, and central processing can take place simultaneously out of different memory banks, or, with certain restrictions, out of the same memory bank. The high-speed and low-speed I/O operations originate, of course, from requests by the central processor unit. The low-speed control unit can service two or more low-speed I/O devices simultaneously, while the high-speed devices can only be operated individually, mainly because their cycle time approaches that of core memory.

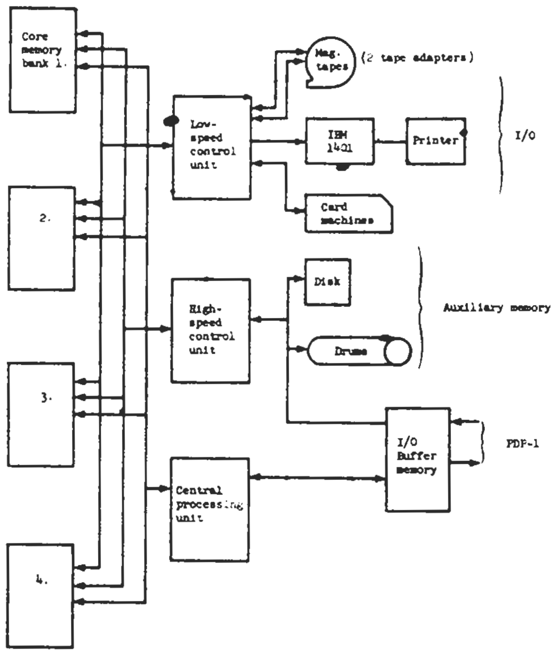

Figure 2. TSS Hardware System.

A memory-protection mechanism and an interrupt, interval (quantum) clock (not shown in the figure) are also integral parts of the TSS computer system. On a bank-by-bank basis only, the memory protection mechanism provides the capability for inhibiting, under program control, the writing of information into one or more memory banks. The quantum clock has the following characteristics:

1. It can be set under program control to a time interval (quantum) anywhere in the range from a few msec. to 400 msec.
2. It can be made to interrupt computer operations after the set interval has elapsed, or after any power-of-two multiple of the set interval (up to a multiple of eight) has elapsed.
3. Under program control, it can be activated and reset.

A summary of the pertinent device characteristics is given in Table I below. The disk file shown in Figure 2 is currently being incorporated into the system.

The Time-Sharing System as it Looks to the User

The time-sharing user today communicates with the Time-Sharing System primarily by means of Teletype. He has at his disposal six basic commands to the system. Briefly, these commands are:

- LOGIN: The user is beginning a run. With this command he gives his identification and a "job number."
- LOAD: The user requests a program to be loaded (currently from tape, eventually, from disk). Once this command is executed, the program is an "object program" in the system.
- GO: The user starts the operation of an object program or restarts the operation of an object program that has been stopped. Once the user gives this com-

Table I. Characteristics of the AN/FSQ-32 Storage Devices

DEVICE	SIZE	WORD RATE	AVERAGE ACCESS TIME
Core Memory	65K	2.5 μsec./wd.	—
Inut/Output Core Memory	16K	2.5 μsec./wd.	—
Magnetic Drums	400K	2.75 μsec. /wd.	10 msec.
Disk File	4000K	11.75 μsec./wd.	225 msec.
Magnetic Tapes	16 Drives	128 μsec./wd. (High density)	5 to 30 msec. (no positioning), depending on whether the tape is at load point, and whether it is being read or written.

mand, he can send Teletype messages to either his object program or the Time-Sharing System.

- STOP: The user stops the operation of an object program.
- QUIT: The user has finished a particular job. Upon receipt of the QUIT, the Time-Sharing System punches a card with certain accounting information on it and removes the object program from the system.
- DIAL: The user may communicate with other users or the computer operators with this command.

In addition to these basic commands, the user has available to him a variety of on-line program debugging, or checkout, functions which give immediate access to any part(s) of an object program.

Briefly, these debugging functions include:

- Open: Displays the contents of the given memory or machine register and uses this as a base address for other debugging commands.
- Modify open register address: Changes the address of the opened register by the given increment or decrement.
- Insert: Inserts the given value into the opened register.
- Mask: Inserts values by the given mask.
- Mode: Displays values according to specified mode (floating, decimal, octal, Hollerith).
- Break point: When a specified point in the program is reached, notifies the user, and (on options) displays registers, and stops or continues the program. As many as five break points are allowed simultaneously.
- Dump: Dumps a given set of registers, either on Teletype or tape.

The actual commands to perform these functions usually include a symbol or address with one or two unique Teletype characters.

Additional Facilities Available to System Users

The commands and devices mentioned so far are facilities available to users or users' object programs directly through the Time-Sharing System's Executive. With these facilities one could run and debug programs that exist in a binary form. To make the system more useful, however, a number of additional devices (called service routines) are available to users. These are themselves run as object programs, so it is clear that there is no limit to the number of service routines that can eventually be made available.

These service routines include programs to file and update symbolic information; compilers; a fancy desk calculator; tape-handling routines; and a number of others including some advanced routines utilizing interpretive techniques. A more detailed description of these interpretive routines appears in the time-sharing applications section, below.

SYSTEM OPERATION AND TIME AND SPACE ALLOCATION

System Operation

The discussion so far has been primarily on the operation of the system from the user's point of view. The following is an over-all description of the system and how it operates.

Basically, the system operates as follows: All object programs are stored on drum, put there as a result of the LOAD command. When a program's time to operate arrives, or, preferably, ahead of this time, it is brought into high-speed memory. If bringing a program into its area in memory causes a storage conflict with another program, the latter must be restored to its place on drums (a process called swapping). A program's turn will end when it initiates an input or output request, when a machine or program error is detected, or when its time is up, the time allotted being determined prior to its turn. At the completion of its turn, its machine environment (e.g., accumulator, index-registers, etc.) is saved, and it either resides in memory until its next turn or is written on drums. This mechanism is controlled by the time-sharing Executive.

As stated before, there is no restriction on the type of object program that can run in the system. Therefore, as much input/output equipment as possible is made available to object programs; thus, object programs may use tapes, displays, and Teletypes for input and output. Other computers can also be treated as input/output devices; further, disk storage is available to object programs. Since it is impractical, in such a system, to have specific Teletypes or tapes referred to by object programs, input/output is done in a general fashion, with all input/output devices given arbitrary names by the object programs and declared to be files used by the object program during its run. Thus, only the Time-Sharing System knows what physical tape drivers, Teletypes, or areas of drums are being used.

The Time-Sharing System's Q-32 Executive occupies 16,384 words of memory, leaving the remainder of memory for object programs. The Executive that exists in the PDP-1 is primarily concerned with maintaining the flow of information to and from the remote devices. It does relatively little decision-making. However, it does determine the kind of input/output device concerned, the type of conversion necessary (if any), and the particular channel of the device with which it is communicating.

The time-sharing Executive in the Q-32 has eight major components. These include routines that perform input/output, perform on-line debugging, interpret commands, assign storage, and schedule object programs. By far the most distinctive feature of the time-sharing Executive, compared to other monitors or executive systems, is the scheduler. Accordingly, a more detailed description of time and space scheduling follows.

Time Allocation and User Capacity

The first problem considered in the Time-Sharing System (TSS) scheduling design was the determination of the minimum amount of time to be given each program during a response cycle of the system. A response cycle is that period of time during which all active programs (i.e., programs requiring central processing time) are serviced. Clearly, to satisfy TSS objectives, this quantum of time (q) must be at least as great as the average amount of time required by an object program to produce a response. Here, of course, we refer only to those programs designed to communicate with a user station (display or keyboard device), and to those programs for which a fast response is desired and can reasonably be expected. In other words, a user requesting a matrix inversion will (and must) expect to wait considerably longer than a user wishing only to see the contents of some register in his program.

Initially, it was obviously not possible to determine *a priori* the distribution of object-program operating times, nor was it even possible to define or classify the group of users requiring these data. The currently available information regarding user programs, and, to some extent, the experience of others, indicated that a q of 50 msec. was sufficient. The extensive recording now being performed during TSS operation is accumulating data that will much more accurately indicate the necessary q size.

In the following section, "worst-case" situations are being treated. "Worst-case" situations are being treated because they, by definition, give the overload threshold or capacity of the system; because they simplify the problem of having to cope with the distributions of object-program sizes and operating times; and because TSS will be operating at, or near, capacity for a high percentage of the time, if the present rate of usage continues. In some cases the "worst-case" values that are used had to be estimated. There is considerable evidence, however, to support the estimates given in the following approximation of the maximum number (n_{max}) of active users that can be serviced in one response cycle, when given the size of the response interval (t_r), the quantum size, and the hardware constraints.

In the current version of TSS the "worst-case" response cycle consists of the following recurrent, non-overlapping sequence of operations: dumping of the last program operated; loading of the next program to operate; allocation of the time interval for operation. For the values of q and t_r that are of interest, the number of active programs in the system can be much larger than the number of memory-protected programs that can be held in core memory at one time; therefore, the above sequence will virtually always be necessary for the operation of each program.

Assuming (as is presently the case) that object programs are not relocatable, we have (in view of the regular, cyclic operation of TSS) the following simple relation,

$$n_{max} = \frac{t_r\ (1 - \eta)}{2t_s + q} \qquad (1)$$

where t_s represents an average value for the time it takes to transfer a program from drum storage to core memory or vice versa, and η is the fraction of time (overhead) used by the Executive during each response cycle.

The fraction of overhead (η) is a difficult quantity to evaluate, and it depends to some extent on n_{max}. Because of the complexity of TSS operation, it is also difficult to estimate η through recording during TSS operation. From experience to date with the system, it is estimated that π ranges from two per cent to fifteen or twenty per cent depending on existing circumstances.

Equation (1) shows that, without major revisions in hardware, a significant inprovement in n_{max} can be achieved only through a decrease in the quantity ($2t_s + q$). In particular, if object programs can be made dynamically relocatable, this quantity can be reduced to the value of $2t_s$ alone. Clearly, this is the best one can do, simply because the speed of the high-speed I/O section in swapping programs in an uninterrupted sequence represents a fundamental upper bound on TSS capacity. Further improvement necessitates an extensive increase in core-memory size, so that at least some active programs can remain in memory during consecutive response cycles. An increase in n_{max} brought about by an increase in the speed of the high-speed I/O section is not economically feasible as can be seen from the equipment description given earlier.

Assuming dynamic relocatability of programs, equation (1) changes to:

$$n_{max} \leq (1 - \eta)\, t_r / 2t_s \qquad (2)$$

In practice, the extent to which the optimum is attained depends on the distributions of object-program sizes and operating times. If $2t_s$ is substantially larger than q, Equation (2) can, for all practical purposes, be considered an equality. Relocatability at load time would, of course, also significantly increase n_{max}, but the improvement that could be expected would be substantially less than that given in Equation (2). For a more specific evaluation of the improvement, a knowledge of the distributions just mentioned is necessary.

The linear relationships between n_{max} and t_r given by Equations (1) and (2) are shown graphically in Figure 3 with the following parameter values: q = 50 msec., η = 0.20, and

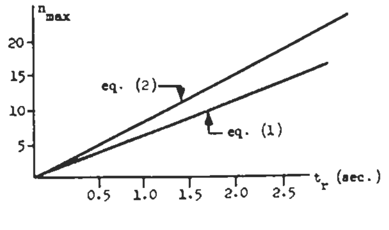

Figure 3. Response Time vs. Number of Users.

$2t_s = 100$ msec. (corresponding to a program size of 16K words). At present, a value of 2.0 sec. is being used for t_r, with a resulting n_{max} of about 11.

Up to now, the efficiency of central processor utilization (relative to unoverlapped I/O time) has been considered of secondary importance, providing that user requirements have been met. Admittedly, this computational efficiency[3] is rather low in the "worst-case" situations. As will be seen in the next section, however, the way in which object programs are sequenced tends to maximize this efficiency for any given load situation. Clearly, the installation of dynamic relocatability in the system would allow an efficiency up to 50% since q can be made equal to 2t, without affecting Equation (2).

It should be emphasized that n_{max} does not represent the maximum number (N_{max}) of user *stations* that can be active at one time; it represents only the maximum number of user *programs* that can be serviced in a fixed response interval under the assumptions given earlier. It has been conservatively estimated that the associated object program is in need of central processor time only ten to twenty per cent of the time during which a user station is in use. Accordingly, it may be possible to make N_{max} considerably larger than n_{max} without significantly jeopardizing user-response requirements. Three important factors figure in the estimate of $\rho = n_{max}/N_{max}$:

1. Relative to computer processing speeds, many applications (e.g., debugging, gaming) consume considerable user time in thinking and output analysis.
2. The average user is less than professional in his use of input devices. A slow manual-input rate, coupled with occasional typing or format errors, will certainly tend to make ρ small.
3. Generally, computer output to user stations takes as much as one to ten seconds.

The estimate of ρ given above was based on the observation of these three factors during system operation and has been justified by the results of the limited amount of recording currently available. In obtaining the precise distribution of the quantity ρ it will be possible to determine the probability of overload for a given N_{max}, or to determine the N_{max} necessary for a given probability of overload. It should also be pointed out that, ultimately in TSS, as in a telephone exchange, several more user stations may be allowed than can actually be in use at one time. The extent to which N_{max} can be exceeded must again be determined by a distribution obtained in the same manner as for ρ.

Sequencing and Priorities

The sequence in which object programs are allocated time is determined by a priority scheme that favors the smaller programs that do not use low-speed I/O time. The amount of time allocated is given by the total time available (t_r), divided by the current number (n) of object programs requesting central processing time. When $n = n_{max}$, the time allocated is given by the minimum quantum discussed in the previous paragraphs.

The priority scheme was adopted to prevent low-speed I/O that was initiated by object programs from degrading the response of those users not using low-speed I/O. Users whose programs require low-speed I/O must expect poorer response, not only because of the low-speed operations, but also because of possible conflicts in object-program I/O requests. Each object program in the system receives a priority according to the criteria in Table II.

Table II. Priority Criteria

PRIORITY	PROGRAM CHARACTERISTICS
1	Program is less than 16K and does not use low-speed I/O.
2	Program is less than 32K and uses low-speed I/O or, program is between 16K and 32K and no low-speed I/O.
3	Program in excess of 32K.

During any given interval of time, Priority 1 programs will receive service first; Priority 2, second; and Priority 3, last. To prevent degradation of response by low-speed I/O, main

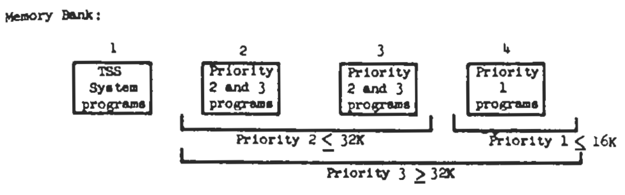

Figure 4. Main Memory Allocation.

memory is partitioned and allocated as shown in Figure 4. Because of the relatively small number of current TSS users, this storage allocation procedure has not yet been imposed on object programs. In the future when the number of Priority 2 and 3 users begins to cause a significant degradation in Priority 1 response, this scheme will be fully implemented as described.

Figure 4 shows that Priority 1 and 2 programs can be multiplexed, but Priority 3 programs preempt practically the entire machine. The priority scheme cannot solve the problem arising when a Priority 3 program undertakes a lengthy, low-speed I/O transfer. The majority of programs using low-speed I/O, however, concern tape transfers, which involve no searching, that take from 50 to 75 msec.

When a program completes operation prior to the expiration of its time allocation for any of the reasons given in the second paragraph of this section, the remaining time will be redistributed among the remaining users requesting service. As a result, the large Priority 2 and 3 users will generally receive more time than the Priority 1 users, thus increasing the potential utilization of central-processor time.

Space Allocation

Although the timing and speed limitations on TSS capability have been of concern, storage limitations are presently far more severe. Storage limitations can be largely removed, however, by acquisition of additional drum space up to the maximum of about 600K. Figure 5 gives a rough idea of how much drum storage must be provided for object programs, to achieve a balance between the speed and capacity of the system. The curves are obtained by letting $n_{max} = \rho N_{max}$ in Eq. (1), $t_r = 2.0$ sec., $\eta = 0.2$, and $\rho = 0.2$.

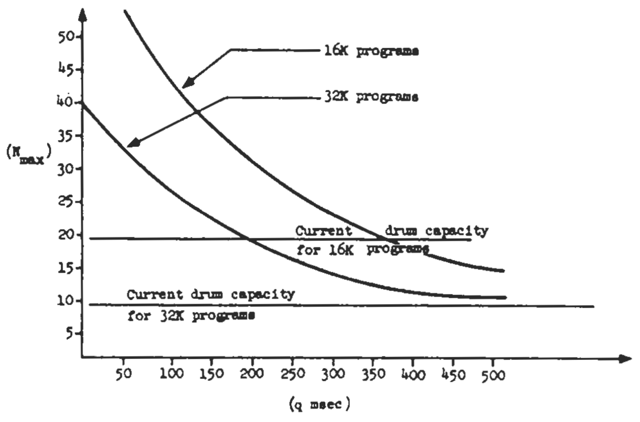

Figure 5. $N_{max} = \frac{1-\eta}{\rho} \cdot \frac{t_r}{2t_s + q}$.

In the initial TSS model (with only eight TTY users), auxiliary memory drum storage was partitioned and allocated in a fixed manner to provide an early working model of the system. This technique proved quite satisfactory at the time, but the number of input stations has now increased to about 48. To accommodate the additional users, a more efficient use of drum storage was necessary. The present method meets this requirement by allocating storage in a contiguous, "head-to-tail" fashion. The adapatability of this storage-allocation method requires searching an inventory of available drum space each time a new program enters the system, and periodically redistributing drum space to maximize the available amount of contiguous drum space. A possible disadvantage of this method is the additional overhead produced, especially when programs must be reshuffled to allocate a sufficient amount of contiguous drum space for a new program. Here again, the performance of this storage allocation technique must be evaluated by statistical recording, since the performance depends strongly on the distribution of program sizes, and on the rates at which programs enter and leave the system. However, at present (and in the foreseeable future) the above rates are so low that the additional overhead produced is negligible.

Future Improvements in TSS Scheduling

There are many ways, including both hardware and software additions to the system, in which the capacity and scheduling efficiency of

TSS can be enhanced. The more or less obvious hardware improvements include:

- Additional core memory
- Additional drum memory
- Relocation mechanism
- Disk storage

The effects of a relocation mechanism and additional drum storage have been described in the previous sections. Additional main memory can be expected to allow for a larger Executive system, larger object programs, and greater scheduling efficiency. However, a substantial improvement in scheduling efficiency must be predicated on the existence of a relocation mechanism, when one makes the obvious assumption that the memory size is small compared to the total size of all active object programs.

The disk file, which is just now being installed and checked out, will supplant tapes for all those applications in which disks are faster and use less machine time. It is expected that disks will be used to store a program library for TSS usage and to store large data bases for object program usage. It is conceivable that disks will also be used for program swapping. The first use of the disk file promises to eliminate a high percentage of manual operations associated with program loading from tapes, and thus to reduce greatly the corresponding delay experienced by users. The second use should save considerable time for the user whose application involves searching through large data bases.

Although the estimates given in this paper are based realistically on current experience, it is not unlikely that user characteristics will evolve quite differently than predicted. Program sizes and/or operating times may grow to a point that invalidates the "worst-case" figures given in this section. It is possible, however, to counteract a certain amount of this degradation by certain improvements in scheduling logic. One improvement would be obtained by taking advantage of the fact that a fairly large class of users exists for whom responses substantially greater than one or two seconds are quite acceptable. In short, it is possible to assign response levels to each user and to service each user just frequently enough to ensure his level of response. Furthermore, the disk file can be used for swapping those programs for which short responses are not necessary. Provided that disk access is in parallel with other high-speed I/O activities, the effective swapping speed can retain the same order of magnitude as for drums.

There are many programs that do not alter themselves during their execution. Thus, as another software improvement, these programs could be treated by the system in two sections: an instruction section and an environment (data) section. During a program's execution it would never be necessary to write the instruction section back on drums; only the environment section and the machine conditions at interrupt would be written back on drums. These and other improvements to TSS are under present investigation. Of principal concern in the investigation of these system changes is the amount of overhead they produce. In some cases the increase in overhead exceeds the expected "improvement" in operating speed and efficiency.

TIME-SHARING APPLICATIONS

To illustrate the "general purpose" nature of the Time-Sharing System, we focus on two interesting programming systems currently operating on TSS as service systems for the user. The first, IPL-TS, is a complete list-processing system for the Information Processing Language V developed by Newell, Simon, and Shaw.[12] The second, TINT, is an on-line *T*eletype *INT*erpreter for the JOVIAL algebraic language developed by SDC.[16, 17] When the Time-Sharing System is equipped with these two programming-language systems, the user is immediately provided with a familiar programming system to ease his transition to programming for time-sharing, and allowed to use, with little or no modification, any code he may have previously written in IPL-V or JOVIAL for other machine systems.

IPL-TS Description

IPL-TS executes interpretively IPL-V code written in accordance with the latest published IPL-V conventions.[13] Some exceptions are noted, particularly in the IPL-TS I/O conventions dictated by machine limitations and time-sharing procedures. More significant, however, are the extensions provided by IPL-TS in the areas of mode of code execution, and improved on-line communication.

IPL-TS can operate in one of two modes at the programmer's option: the "production" mode or the "debugging" mode. The production mode is designed for maximum code execution, and is used essentially for checked out code.

Though code is still executed interpretively, a suppression of all debugging functions in the production mode has produced a four-fold increase in execution rate over debugging-mode operation. Execution rates of over 400,000 cycles per minute, which compare favorably with other non-time-sharing IPL systems, are common. To the IPL-TS user, production-mode operation is analogous, as we shall see later, to the TINT user compiling his checked-out code with the Time-Sharing JOVIAL Compiler (JTS). Debugging-mode operation, on the other hand, is designed for maximum user efficiency and greater on-line programmer control over the execution of his program. The debugging mode allows all the standard IPL options; it also permits a number of on-line functions not common to IPL systems. These include:

1. Optional breakpoint action at any monitor point, whereby the currently executing program is suspended until completion of the execution of any on-line, programmer-specified routine;
2. On-line, symbolic program composition and/or debugging;
3. Optional automatic or on-line programmer-controlled execution of a full "back trace" routine that prints up to the last 100 interpretation cycles. This routine is executed by IPL-TS automatically at each system-detected error occurrence as a debugging diagnostic; and
4. A flexible, "thin skinned" system error trap mechanism permitting programmer specification of trapping actions for all system-detected errors.

TINT Description

TINT is a two-pass interpretive program system for time-sharing use, and operates upon a subset of the JOVIAL problem-oriented language.

TINT includes a generator, a set of operator subroutines, and the interpreter. The generator was acquired from a current IBM 7090 JOVIAL compiler and was modified to handle the particular JOVIAL dialect used by TINT. The operator subroutines and the interpreter are original code developed specifically for TINT.

The generator (first pass) scans the input JOVIAL statements and translates them into an intermediate Polish prefix language. Grammer checking is performed during the translation. The language subset allowed may include the arithmetic, relational, and Boolean operators; procedure calls; data table, array, and item (integer, floating point, and Hollerith) declarations; and the GOTO, IF, STOP, READ, and PRINT statements. The READ and PRINT statements were added to the language specifically for time-sharing operation.

Operator subroutines comprise the primitive functions used by the interpreter to perform the actions specified in the intermediate language. The interpreter (second pass) scans the intermediate language for the current operator prefix and its arguments, and executes the corresponding operator subroutine that computes on these arguments.

The TINT user is permitted a number of options in composing and executing his code. He may reference code stored in a binary library tape of his own composition; he may file away any current code on tape for subsequent use, or for compilation with JTS after the code has been exercised and debugged; and, he may optionally execute code from a prestored tape or from the Teletype.

On-Line Program Composition

Both IPL-TS and TINT allow the user to write symbolic programs on-line and to execute them immediately, by themselves or in conjunction with previously coded routines. With IPL-TS, the programmer uses the special system routine, Linear IPL (LIPL),* which accepts

* LIPL was designed and coded by R. Dupchak while consultant to the RAND Corporation, Santa Monica.

```
LIPL READY!

RT ACKTEST=(AØ A1 A2 A3,ACKTES()
    AØ=(J154 1Ø$CYC.D$ J157 1Ø$H3=$ J157 1Ø+1Ø  J161
      1Ø$SEC.D$ J157 1Ø$TIME=$ J157 1Ø+1Ø J161
      1Ø$ACK($ J157 1ØØØ J157 1Ø$,$ J157 1ØNØ J157 1Ø$)=$ J157)

' AØ SETS UP PRINT LINE '

     A1=(1ØH3 J12Ø J5Ø *$1)
' A1 SAVES H3 COPY IN WØ, AND CLOCK IN $3ØØ '

     A2=(1ØMØ J117 1ØNØ 7Ø9-1 J125,JØ) 9-1=(J117 7Ø9-2 1ØNØ J125,9-3)
    9-2=(1Ø+1 1ØNØ J111 A2,9-3 5ØK1 1ØMØ 1ØMØ J111 A2 J125,JØ)

' A2 COMPUTES AC(MØ,NØ) '

     A3=(1Ø+9 J16Ø 1ØH3 11WØ 11WØ J111 J157 1Ø+29 J16Ø *$3 1Ø+49
       J16Ø 1ØNØ J157 J155 11WØ J9,J3Ø)

' A3 COMPUTES DELTA  H3 AND DELTA TIME. ALSO SETS VARIABLES IN
 PRINT LINE AND PRINTS LINE '

     MØ=+2 NØ=+1 K1=+1   ' INITIAL VALUES ' NL  GT ACKTEST
CYC.DH3=17Ø        SEC.DTIME=Ø.24Ø7    ACK(2,1)=5
CYC.DH3=11Ø2       SEC.DTIME=Ø.7929    ACK(2,5)=13
CYC.DH3=5254       SEC.DTIME=3.4Ø91    ACK(2,13)=29
CYC.DH3=25294      SEC.DTIME=16.3442   ACK(2,29)=61
CYC.DH3=19Ø6Ø7     SEC.DTIME=121.Ø779  ACK(2,61)=125

!STOP
```

Figure 6. Typescript of Ackermann's Function.

IPL code on-line in a symbolic, linear, parenthesis format convenient for keyboard input. Figure 6 presents an example of LIPL being used to compose and execute Ackermann's function[8] on-line. TINT, which was developed specifically for on-line program composition, accepts JOVIAL statements on-line in the same linear format used for compiler input.

The ability to program on-line frees the programmer from having to concern himself with all the formalities of punched card accounting. With experience and facility, he programs on-line directly from his thoughts or, for more difficult problems, directly from a flow diagram, circumventing such time-consuming tasks as program-coding-sheet preparation, key punching, card sorting, editing, and prestoring. The time saved by the programmer can be applied to other coding tasks or to quality review of his current code.

No programmer, of course, could compose a large program at one sitting with either of these systems, but this is a human, not a system, limitation; LIPL has no upper bound, and TINT's 600-statement limit effectively exceeds a human's short-term comprehension. Optimally, these systems should be used for programs that can be written and debugged in one or two sittings (usually under 100 IPL instructions or 50 JOVIAL statements).

There are three immediate consequences of this practical size limitation. First, many nontrivial, one-shot programs, such as for statisti-

```
LOGIN Ø173 JDX.25
$OK LOG ON 14
LOAD TINT 1796
$WAIT

$LOAD OK
GO
 $MSG IN

            START   "BEGINS NEW PROGRAM"
           ITEM N F $   "NUMBER OF CASES"
           ITEM SUMX F $    "SUM OF VALUES"
           ITEM XBAR F $    "ARITH. MEAN"
           ITEM SDEV F $    "STD. DEVIATION"
           READ  N, SUMX $
           XBAR = SUMX/N $
           PRINT 6H(MEAN =), XBAR $
           SDEV = ((SUMX**2.Ø-N*XBAR**2.Ø)/(N-1))**Ø.5 $
           PRINT 8H(S. D. = ), SDEV $
           TERM $    "CAUSES EXECUTION OF PROGRAM"

               N = ? 12.Ø

               SUMX = ? 1478.Ø

            MEAN =     123.2

            S. D. =      1Ø.3

           ILT EXECUTION COMPLT

$QUIT
 $MSG IN
```

Figure 7. Example of the Use of TINT as a "Desk Calculator."

cal computations, can be coded, debugged, and executed at one sitting. Often a programmer himself will refrain from writing such programs, knowing the time and effort involved. Figure 7 shows the Teletype communication resulting from an exercise using TINT as a "desk calculator" for computing the standard deviation of a set of research data. Second, large programs take on a modular structure; that is, large programs become a concatenation of numerous smaller programs and subroutines. Third, programmers begin to amass personal libraries of short utility subroutines, which they use to build larger programs. Clearly, consequences two and three would not exist, except in trivial cases, if it were not possible to work one day with code developed on prior days. Both IPL-TS and TINT provide this capability.

TINT may accept symbolic input from magnetic tape, and can integrate this input with on-line Teletype input when so directed by the user. Thus the results of one day's coding can be filed on tape for later use. An alternative, if the symbolic JOVIAL statements have been executed and debugged, is to compile the code and save the binary output on a binary library tape, thus, again, integrating previous work with current code; however, the binary library approach has greatest value when used for utility routines.

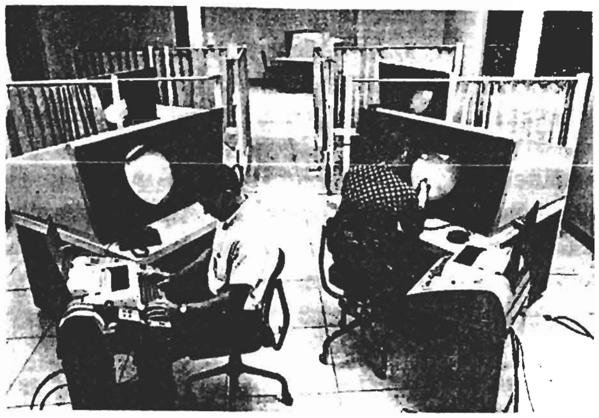

Figure 8. Accessing the Computer with Model 33 Teletypes and Displays.

IPL-V is essentially a language of subroutines (composed from an inventory of some 200 system subroutines called J routines or primitives). Programs written in IPL-V are usually modular hierarchies of subroutines. Therefore, on-line composition of IPL-V programs is a natural extension of the language, and many alternatives for continuity of programming across many days of operation already exist within the language. For example, the programmer may "fire" a J166 (Save For Restart) at any time and continue from that point at a later date, or he may load a program from symbolic tape using the loader or J165 (Load Routines and Data) and continue using LIPL on-line.

Therefore, the attributes of IPL-TS and TINT, when combined with a programmer's imagination and skill during on-line program composition, reduce significantly the tedious, uncreative tasks of code preparation and increase productivity. This point is particularly apparent to all programmers who have been required to debug code that they wrote several days earlier, and that has grown "stale" while it was being keypunched, compiled, and executed. Instead of expending additional time and energy becoming reacquainted with his code before he can correct his errors, the programmer can, by composing the code on-line and executing it immediately, debug while the code is still fresh in his mind.

On-Line Program Debugging

The particular ability of IPL-TS and TINT to detect, locate, and correct program errors on-line is perhaps their greatest asset, since it leads to substantial decrease in program turnaround time. In effect, IPL-TS and TINT increase the programmer's debugging efficiency by allowing him to check out more code per day than would be possible with non-time-sharing operation.

Error Detection is the first step in debugging any program. Errors may be classed as either grammatical errors in language or format, or logical errors in code execution. The generator screens out most grammatical errors for TINT, and either the loader or LIPL performs the same task for IPL-TS. Logical-error detection, however, is a more difficult task, even with IPL-TS and TINT. The advantage of these systems for error detection is their responsiveness to the programmer. He may choose to develop on-line, special-purpose debugging tools to suit his individual preference, or he may use those debugging tools provided by the system. For example, IPL-TS currently provides an error trap for some twenty illegal IPL operations resulting from faulty program logic; when such errors occur, IPL-TS attempts to provide the programmer with as much information as possible to help him correct his error. First, an error message is sent to the programmer to inform him of the error's occurrence and of its nature. Second, a special system routine, Trace Dump (discussed below), provides him with a "back trace" of the code leading up to the error to help him locate the cause of the error. Finally, the system pauses at a breakpoint, to allow him time to correct the error. However, all three steps may be altered, since the IPL-TS error trap mechanism is designed with a "thin skin" to allow the programmer to substitute his own trapping action in lieu of that provided by the system.

With TINT, logical-error detection is left more to the imagination of the programmer. TINT allows the programmer to insert a PRINT statement, with numerous item names as arguments, at any point in his program. When it encounters this statement during program execution, TINT responds by printing on the user's Teletype the current values of all specified items. In this fashion, the programmer may take item snapshots at critical points in his program. The power of the PRINT statement for logical-error detection is amplified when combined with the TINT READ statement. The READ statement is the converse of the PRINT statement. When TINT encounters this statement during program execution, the programmer must insert the current values of prespecified items. By judicious use of the READ and PRINT statements, the programmer can repeatedly exercise a program with different initial conditions and review his results with input/output transfer-function analysis.

Thus, on-line user-program communication increases a programmer's debugging efficiency by increasing his ability to detect program errors. It is typical for a programmer, checking out new code with IPL-TS or TINT, to detect and correct half a dozen program errors in the first hour of operation; such error correction might easily have required a week with conventional programming systems.

Error location, the pinpointing of the erroneous code, is often considered no different from error detection. This may be true for grammatical errors, but is far from true for logical errors. The knowledge that an error exists does not, in and of itself, narrow the search for the error's location. The user of IPL-TS, therefore, is provided with a description of the system-detected error and the aforementioned back trace of the code leading up to the error. Back tracing by the system is performed in the debugging mode by the special system routine Trace Dump, which prints a full trace of up to the last 100 interpretation cycles, in reverse order (last cycle first). The number of previous cycles printed is controllable on-line. Experience shows that the location of an error can usually be found within the first five cycles printed, and that it is rarely necessary to go deeper than ten cycles back. For logical errors not detected by the system, the programmer has available all the standard IPL-V Monitor Point functions; in addition, IPL-TS extends these functions to include breakpoint operation as a programmer-initiated option. The option may be invoked at load time or during program execution. In addition, the IPL primitive J7 (Halt) has been implemented as an alternative breakpoint mechanism. When a breakpoint is encountered by IPL-TS, the programmer is notified and requested to enter the name of any regionally defined routine, which is then executed immediately. Upon completion of the routine, the programmer is again queried. He may continue to fire routines at the breakpoint, or he may exit back to the prior program, the context of which has remained undisturbed.

Breakpoints are not a panacea for locating erroneous code; however, they do provide additional control flexibility at critical points in a program. In fact, the user of TINT must rely almost exclusively on breakpoint logic for locating erroneous code: the aforementioned READ and PRINT statements are in effect breakpoint statements. For elusive errors these statements may be used to bracket groups of JOVIAL statements, and in extreme cases, individual JOVIAL statements. TINT also provides a STOP statement, which is also a breakpoint statement. When the interpreter encounters the STOP statement, the program is suspended until directed by the user to continue. The user may also reexecute his program from a STOP breakpoint, or he may enter new code or edit prior code before continuing. TINT's STOP statement is analogous to the IPL-TS J7 (Halt) primitive.

Error correction in symbolic code with either IPL-TS or TINT is essentially on-line program composition. LIPL allows the IPL programmer to erase, extend, or modify selectively any user routine existing in the system. TINT, similarly, allows the programmer to edit any JOVIAL code written, on a statement-by-statement basis.

Here, again, the programmer's control over his program is effectively increased. He can correct code in several minutes instead of the several days typical with most computer installations.

SUMMARY AND CONCLUSION

There are some obvious advantages to this kind of system that have been borne out in practice. There is a large class of problems whose compute time is extremely small in relation to the total time the problem is on the computer. This is because a large percentage of time is taken up by human thought and computer input/output. In fact, the use of a computer for this kind of application in a non-time-sharing mode is so inefficient that it would not be worthwhile to run. There are many examples of this kind of problem. The one that most programmers are familiar with is console debugging, that is, the checkout of programs with the programmer at the computer—anathema to most computer managers, but desired by a large number of programmers. These kinds of applications have been run with a high degree of success in this Time-Sharing System, with each person involved actually feeling he has the whole computer to himself.

At the other end of the spectrum are those programs that compute for essentially one hundred per cent of the time they are on the computer. If these programs compute for long periods, say a matter of minutes, they will completely usurp their allotted time and thus tend to make the on-line user wait for the maximum response period possible. Time-sharing does not benefit this kind of user, except that this kind of program can be run "in the background" while other on-line interaction programs are idle. In the SDC installation, the percentage of these long-period compute programs has been small, so that no serious system response time delays have been noticed from them.

Questions frequently asked are, "Do people like the system?" "Does it produce better results than other, more standard techniques?" Both the questions are difficult to answer in an absolute sense. However, some reasonable observations can be made that apply to this system and probably to others of this kind.

First, those on-line interaction programs that used to run in a non-time-sharing mode but were converted to time-sharing produce results that are as valid as before but with greater efficiency in computer operation, since a number of different ones are run simultaneously.

Next, the on-line debugging capability has proved very valuable. This system of debugging gives a feeling of closeness to the computer and control over the program, so that debugging time is reduced considerably while the efficiency of computer utilization stays high. Also, although the tools available so far have been relatively few and unsophisticated, one can see the advantages to be gained by giving everyone immediate access and response from a computer. "Directed" computer runs are the mode of operation. Every step taken is taken only as a result or verification of the previous step. If things do not go as planned, alternative paths can be followed immediately. Before time-sharing, one had two choices: "submitting" of a run, followed by an anxious waiting period climaxed by a sigh (or worse) and a resubmitting of the same run; or one-man on-line interaction with the computer, which benefitted that person, but caused consternation on the part of others waiting for computer runs.

This kind of system must be made foolproof. Due to the nature of this system, one must have a reasonably long time of uninterrupted operation to get satisfactory results. This implies several things:

1. The system Executive must be reliable and able to account for any condition that may arise, including object program and machine errors.
2. The machine must be reliable. Although the system must provide the ability to analyze each computer error and isolate and stop only the particular object program or programs affected, frequent or solid computer errors can cause the entire system and all object programs to terminate.
3. Certain hardware features are essential. These include: *Memory protection*—the ability to prevent object programs from destroying each other or the Executive system; and *high-speed large-storage random-access devices*—the major bottleneck in a system of this kind is the slow rate at which object programs can be moved in and out of memory. Also, the use of magnetic tapes for such functions as the permanent storage of programs and data files creates operational and timing problems that can be overcome with the use of large drums or disks; also essential is *clock interrupt capability*—the system requires that no single program run for an excessively long time.

Therefore a clock that can be set to interrupt operation at various intervals is necessary for complete control and the assurance of adequate response time.

When this Time-Sharing System first became operational, it had no memory protection, its Executive was unreliable, and its computer was beset by a much heavier load than it was used to and reacted accordingly. With these obstacles, the early users were subject to frustrations unlike many found in the twentieth century. The system's life expectancy was no more than ten minutes. The only remarkable thing about the early months was that anything useful was accomplished. Interestingly enough, however, some work was accomplished, primarily through patience on the part of the users. With the passage of time, many of the problems have been alleviated through both equipment and programming improvements, so that now the system runs with considerably more continuity and reliability.

Since the system became operational, it has been used in a wide variety of applications. These applications have, for the most part, been checked out using the Time-Sharing System and have been run productively during time-sharing. Some of the specific applications for which time-sharing has been used are:

- Natural Language Processors—used for parsing English sentences, answering questions, and interpreting sentence-structured commands.
- Group Interaction Studies—in which teams or players are matched against each other and the computer is used to measure individual and team performance.
- General Display Programming—in which the programs are used as vehicles for generating and modifying visual displays according to the users' keyboard inputs.
- A FORTRAN-to-JOVIAL Translator—symbolic JOVIAL program tapes are produced for FORTRAN tape inputs.
- Simulated Alternate Mobile Command Post—a realistic simulation of the A.M.C.P. has been produced, and the display requirements for this organization are studied within this framework.

Of course, a number of other routines, games, and services have been and are being developed under the system.

One of the "disadvantages" in using a time-sharing system such as this is the fact that most computer runs require the presence of one or more people. Users of many large-scale computers are accustomed to remaining detached from the actual computer runs and are sometimes reluctant to follow the runs closely. However, the elapsed time for completing jobs using these "on-line" techniques is normally dramatically reduced compared to a more remote operation, and this reduced time has been noted in the use of the time-sharing system.

It is interesting to watch a group of people using a computer simultaneously but solving different problems using different tools. At the computer console itself, one can usually see all the available tape drives busy, typewriters busy, drum indicators indicating the drums are busy, the punch punching, on occasion, and the card-reader going at anywhere from quarter to full speed. For those who judge the worth of a computer by the amount of equipment used per second, time-sharing is well worth its investment.

Since the system has been under development (it was begun in January 1963), the number of existing services has been expanding rapidly. One can envision the development of an increasing number of on-line programming aids and techniques of utilizing keyboards, displays, and groups of computers to make a time-sharing network a truly powerful device.

It is certainly conceivable that, in the not too distant future, many people will have at their fingertips a device that, at a reasonable cost, enables them to enter an operating network such as this one. While in this network, they will have access to routines, techniques, and computing power unavailable to them by other means. The computing power will include not only the Executive computer but the other computers that are in the network as well. Thus, the possibility of large-scale time-sharing networks seems to be one of the more promising developments in computer technology today.

BIBLIOGRAPHY

The following represents a collection of general as well as source material.

1. BOILEN, S. "How the Time-Sharing System Looks Now," Cambridge, Massachusetts, Bolt, Beranek and Newman, Inc. (Unpublished Memo. TS-3), April 2, 1962.
2. COFFMAN, E. G. *A General Flow Chart Description of the Time-Sharing System.* SDC TM-1639/000/00, December 12, 1963.
3. CORBATO, F. J., and others. *The Compatible Time-Sharing System. A Programmer's Guide.* Cambridge, Massachusetts, M.I.T. Press, 1963.
4. CORBATO, F. J., M. MERWIN-DAGGETT, and R. C. DALEY. "An Experimental Time-Sharing System," *Proceedings of the Spring Joint Computer Conference.* 1962, pp. 335-344.
5. FREDKIN, E. "The Time-Sharing of Computers," *Computers and Automation.* v. 12, November 1963, pp. 12-20.
6. GALLENSON, L. *On-Line I/O Processor for the Command Research Laboratory. The PDP-1-C-30.* SDC TM-1653, December 23, 1963.
7. KEMPER, D. A. *Operation of CRL Teletype System.* SDC TM 1488/000/00, September 18, 1963.
8. KLEENE, S. C. *Introduction to Metamathematics.* Van Nostrand, 1952.
9. LICKLIDER, J. C. R. "Man-Computer Symbiosis," *IRE Transactions on Human Factors in Electronics,* V.HFE-1, March 1960, pp. 4-10.
10. LICKLIDER, J. C. R., and W. E. CLARK. "On-Line Man-Computer Communication," *Proceedings of the Spring Joint Computer Conference,* 1962, pp. 113-128.
11. McCARTHY, J., S. BOILEN, E. FREDKIN, and J. C. R. LICKLIDER. "A Time-Sharing Debugging System for a Small Computer," *Proceedings of the Spring Joint Computer Conference,* May 1963, pp. 51-57.
12. NEWELL, A. (Ed.) *Information Processing Language V Manual.* Englewood Cliffs, N.J., Prentice-Hall, Inc., 1961.
13. NEWELL, A. (Ed.) *IPL-V Programmer's Reference Manual.* RAND Corporation, RM-3739-RC, June 1963.
14. ROSENBERG, A. M. *Externally-Generated Priority Assignment for Program Operation in the ARPA-SDC Time-Sharing System.* SDC TM-1159/000/00, April 8, 1963.
15. ROSENBERG, A. M. (Ed.) *Command Research Laboratory User's Guide.* SDC TM-1354, November 19, 1963.
16. SHAW, C. J. "JOVIAL," *Datamation* 7, 6 (June 1961), pp. 28-32.
17. SHAW, C. J. "Programmer's Look at JOVIAL in an ALGOL Perspective," *Datamation* 7, 10 (Oct. 1961), pp. 46-50.
18. SLAYBAUGH, J. A. *The AN/FSQ-32. A Description and Coding Manual for Experienced Programmers.* SDC TM-1489/000/01, December 1963.
19. STRACHEY, C. "Time-Sharing in Large Fast Computers," *Proceedings of the International Conference on Information Processing,* Paris, UNESCO, 1960, pp. 336-341.
20. WEISSMAN, C., and M. KAHN. *IPL-TS Programmer's Reference Manual.* SDC TM-1581/000/00, December 16, 1963.