Fernando J. Corbató, Marjorie Merwin Daggett, Robert C. Daley, An Experimental Time-Sharing System, In Proc. AFIPS 1962 SJCC, Vol. 21, Spartan Books, New York, 335-344.

AN EXPERIMENTAL TIME-SHARING SYSTEM

Fernando J. Corbató, Marjorie Merwin-Daggett, Robert C. Daley
Computer Center, Massachusetts Institute of Technology
Cambridge, Massachusetts

Summary

It is the purpose of this paper to discuss briefly the need for time-sharing, some of the implementation problems, an experimental time-sharing system which has been developed for the contemporary IBM 7090, and finally a scheduling algorithm of one of us (FJC) that illustrates some of the techniques which may be employed to enhance and be analyzed for the performance limits of such a time-sharing system.

Introduction

The last dozen years of computer usage have seen great strides. In the early 1950's, the problems solved were largely in the construction and maintenance of hardware; in the mid-1950's, the usage languages were greatly improved with the advent of compilers; now in the early 1960's, we are in the midst of a third major modification to computer usage: the improvement of man-machine interaction by a process called time-sharing.

Much of the time-sharing philosophy, expressed in this paper, has been developed in conjunction with the work of an MIT preliminary study committee, chaired by H. Teager, which examined the long range computational needs of the Institute, and a subsequent MIT computer working committee, chaired by J. McCarthy. However, the views and conclusions expressed in this paper should be taken as solely those of the present authors.

Before proceeding further, it is best to give a more precise interpretation to time-sharing. One can mean using different parts of the hardware at the same time for different tasks, or one can mean several persons making use of the computer at the same time. The first meaning, often called multiprogramming, is oriented towards hardware efficiency in the sense of attempting to attain complete utilization of all components[5,6,7,8]. The second meaning of time-sharing, which is meant here, is primarily concerned with the efficiency of persons trying to use a computer[1,2,3,4]. Computer efficiency should still be considered but only in the perspective of the total system utility.

The motivation for time-shared computer usage arises out of the slow man-computer interaction rate presently possible with the bigger, more advanced computers. This rate has changed little (and has become worse in some cases) in the last decade of widespread computer use.[10]

In part, this effect has been due to the fact that as elementary problems become mastered on the computer, more complex problems immediately become of interest. As a result, larger and more complicated programs are written to take advantage of larger and faster computers. This process inevitably leads to more programming errors and a longer period of time required for debugging. Using current batch monitor techniques, as is done on most large computers, each program bug usually requires several hours to eliminate, if not a complete day. The only alternative presently available is for the programmer to attempt to debug directly at the computer, a process which is grossly wasteful of computer time and hampered seriously by the poor console communication usually available. Even if a typewriter is the console, there are usually lacking the sophisticated query and response programs which are vitally necessary to allow effective interaction. Thus, what is desired is to drastically increase the rate of interaction between the programmer and the computer without large economic loss and also to make each interaction more meaningful by extensive and complex system programming to assist in the man-computer communication.

To solve these interaction problems we would like to have a computer made simultaneously available to many users in a manner somewhat like a telephone exchange. Each user would be able to use a console at his own pace and without concern for the activity of others using the system. This console could as a minimum be merely a typewriter but more ideally would contain an incrementally modifiable self-sustaining display. In any case, data transmission requirements should be such that it would be no major obstacle to have remote installation from the computer proper.

The basic technique for a time-sharing system is to have many persons simultaneously using the computer through typewriter consoles with a time-sharing supervisor program sequentially running each user program in a short burst or quantum of computation. This sequence, which in the most straightforward case is a simple round-robin, should occur often enough so that each user program which is kept in the high-speed memory is run for a quantum at least once during each approximate human reaction time (~.2 seconds). In this way, each user sees a computer fully responsive to even single key strokes each of which may require only trivial computation; in the non-trivial cases, the user sees a gradual reduction of the response time which is proportional to the complexity of the response calculation, the slowness of the computer, and the total number of active users. It should be clear, however, that if there are n users actively requesting service at one time, each user will only see on the average 1/n of the effective computer speed. During the period of high interaction rates while debugging programs, this should not be a hindrance since ordinarily the required amount of computation needed for each debugging computer response is small compared to the ultimate production need.

Not only would such a time-sharing system improve the ability to program in the conventional manner by one or two orders of magnitude, but there would be opened up several new forms of computer usage. There would be a gradual reformulation of many scientific and engineering applications so that programs containing decision trees which currently must be specified in advance would be eliminated and instead the particular decision branches would be specified only as needed. Another important area is that of teaching machines which, although frequently trivial computationally, could naturally exploit the consoles of a time-sharing system with the additional bonus that more elaborate and adaptive teaching programs could be used. Finally, as attested by the many small business computers, there are numerous applications in business and in industry where it would be advantageous to have powerful computing facilities available at isolated locations with only the incremental capital investment of each console. But it is important to realize that even without the above and other new applications, the major advance in programming intimacy available from time-sharing would be of immediate value to computer installations in universities, research laboratories, and engineering firms where program debugging is a major problem.

Implementation Problems

As indicated, a straightforward plan for time-sharing is to execute user programs for small quantums of computation without priority in a simple round-robin; the strategy of time-sharing can be more complex as will be shown later, but the above simple scheme is an adequate solution. There are still many problems, however, some best solved by hardware, others affecting the programming conventions and practices. A few of the more obvious problems are summarized:

Hardware Problems:

1. Different user programs if simultaneously in core memory may interfere with each other or the supervisor program so some form of memory protection mode should be available when operating user programs.

2. The time-sharing supervisor may need at different times to run a particular program from several locations. (Loading relocation bits are no help since the supervisor does not know how to relocate the accumulator, etc.) Dynamic relocation of all memory accesses that pick up instructions or data words is one effective solution.

3. Input-output equipment may be initiated by a user and read words in on another user program. A way to avoid this is to trap all input-output instructions issued by a user's program when operated in the memory protection mode.

4. A large random-access back-up storage is desirable for general program storage files for all users. Present large capacity disc units appear to be adequate.

5. The time-sharing supervisor must be able to interrupt a user's program after a quantum of computation. A program-initiated one-shot multivibrator which generates an interrupt a fixed time later is adequate.

6. Large core memories (e.g. a million words) would ease the system programming complications immensely since the different active user programs as well as the frequently used system programs such as compilers, query programs, etc. could remain in core memory at all times.

Programming Problems:

1. The supervisor program must do automatic user usage charge accounting. In general,

the user should be charged on the basis of a system usage formula or algorithm which should include such factors as computation time, amount of high-speed memory required, rent of secondary memory storage, etc.

2. The supervisor program should coordinate all user input-output since it is not desirable to require a user program to remain constantly in memory during input-output limited operations. In addition, the supervisor must coordinate all usage of the central, shared high-speed input-output units serving all users as well as the clocks, disc units, etc.

3. The system programs available must be potent enough so that the user can think about his problem and not be hampered by coding details or typographical mistakes. Thus, compilers, query programs, post-mortem programs, loaders, and good editing programs are essential.

4. As much as possible, the users should be allowed the maximum programming flexibility both in choices of language and in the absence of restrictions.

Usage Problems

1. Too large a computation or excessive typewriter output may be inadvertently requested so that a special termination signal should be available to the user.

2. Since real-time is not computer usage-time, the supervisor must keep each user informed so that he can use his judgment regarding loops, etc.

3. Computer processor, memory and tape malfunctions must be expected. Basic operational questions such as "Which program is running?" must be answerable and recovery procedures fully anticipated.

An Experimental Time-Sharing System for the IBM 7090

Having briefly stated a desirable time-sharing performance, it is pertinent to ask what level of performance can be achieved with existant equipment. To begin to answer this question and to explore all the programming and operational aspects, an experimental time-sharing system has been developed. This system was originally written for the IBM 709 but has since been converted for use with the 7090 computer.

The 7090 of the MIT Computation Center has, in addition to three channels with 19 tape units, a fourth channel with the standard Direct Data Connection. Attached to the Direct Data Connection is a real-time equipment buffer and control rack designed and built* under the direction of H. Teager and his group. This rack has a variety of devices attached but the only ones required by the present systems are three flexowriter typewriters. Also installed on the 7090 are two special modifications (i.e. RPQ's): a standard 60 cycle accounting and interrupt clock, and a special mode which allows memory protection, dynamic relocation and trapping of all user attempts to initiate input-output instructions.

In the present system the time-sharing occurs between four users, three of whom are on-line each at a typewriter in a foreground system, and a fourth passive user of the background Fap-Mad-Madtran-BSS Monitor System similar to the Fortran-Fap-BSS Monitor System (FMS) used by most of the Center programmers and by many other 7090 installations.

Significant design features of the foreground system are:

1. It allows the user to develop programs in languages compatible with the background system,

2. Develop a private file of programs,

3. Start debugging sessions at the state of the previous session, and

4. Set his own pace with little waste of computer time.

Core storage is allocated such that all users operate in the upper 27,000 words with the time-sharing supervisor (TSS) permanently in the lower 5,000 words. To avoid memory allocation clashes, protect users from one another, and simplify the initial 709 system organization, only one user was kept in core memory at a time. However, with the special memory protection and relocation feature of the 7090, more sophisticated storage allocation procedures are being implemented. In any case, user swaps are minimized by using 2-channel overlapped magnetic tape reading and writing of the pertinent locations in the two user programs.

The foreground system is organized around commands that each user can give on his typewriter and the user's private program files which presently (for want of a disc unit) are kept on a separate magnetic tape for each user.

* This group is presently using another approach[9] in developing a time-sharing system for the MIT 7090.

For convenience the format of the private tape files is such that they are card images, have title cards with name and class designators and can be written or punched using the off-line equipment. (The latter feature also offers a crude form of large-scale input-output.) The magnetic tape requirements of the system are the seven tapes required for the normal functions of the background system, a system tape for the time-sharing supervisor that contains most of the command programs, and a private file tape and dump tape for each of the three foreground users.

The commands are typed by the user to the time-sharing supervisor (not to his own program) and thus can be initiated at any time regardless of the particular user program in memory. For similar coordination reasons, the supervisor handles all input-output of the foreground system typewriters. Commands are composed of segments separated by vertical strokes; the first segment is the command name and the remaining segments are parameters pertinent to the command. Each segment consists of the last 6 characters typed (starting with an implicit 6 blanks) so that spacing is an easy way to correct a typing mistake. A carriage return is the signal which initiates action on the command. Whenever a command is received by the supervisor, "WAIT", is typed back followed by "READY." when the command is completed. (The computer responses are always in the opposite color from the user's typing.) While typing, an incomplete command line may be ignored by the "quit" sequence of a code delete signal followed by a carriage return. Similarly after a command is initiated, it may be abandoned if a "quit" sequence is given. In addition, during unwanted command typeouts, the command and output may be terminated by pushing a special "stop output" button.

The use of the foreground system is initiated whenever a typewriter user completes a command line and is placed in a waiting command queue. Upon completion of each quantum, the time-sharing supervisor gives top priority to initiating any waiting commands. The system programs corresponding to most of the commands are kept on the special supervisor command system tape so that to avoid waste of computer time, the supervisor continues to operate the last user program until the desired command program on tape is positioned for reading. At this point, the last user is read out on his dump tape, the command program read in, placed in a working status and initiated as a new user program. However, before starting the new user for a quantum of computation, the supervisor again checks for any waiting command of another user and if necessary begins the look-ahead positioning of the command system tape while operating the new user.

Whenever the waiting command queue is empty, the supervisor proceeds to execute a simple round-robin of those foreground user programs in the working status queue. Finally, if both these queues are empty, the background user program is brought in and run a quantum at a time until further foreground system actively develops.

Foreground user programs leave the working status queue by two means. If the program proceeds to completion, it can reenter the supervisor in a way which eliminates itself and places the user in dead status; alternatively, by a different entry the program can be placed in a dormant status (or be manually placed by the user executing a quit sequence). The dormant status differs from the dead status in that the user may still restart or examine his program.

User input-output is through each typewriter, and even though the supervisor has a few lines of buffer space available, it is possible to become input-output limited. Consequently, there is an additional input-output wait status, similar to the dormant, which the user is automatically placed in by the supervisor program whenever input-output delays develop. When buffers become near empty on output or near full on input, the user program is automatically returned to the working status; thus waste of computer time is avoided.

<u>Commands</u>

To clarify the scope of the foreground system and to indicate the basic tools available to the user, a list of the important commands follows along with brief summaries of their operations:

1. | α

   α = arbitrary text treated as a comment.

2. login | α | β

   α = user problem number
   β = user programmer number

   Should be given at beginning of each user's session. Rewinds user's private file tape; clears time accounting records.

3. logout

   Should be given at end of each user's session. Rewinds user's private file tape; punches on-line time accounting cards.

4. input

   Sets user in input mode and initiates automatic generation of line numbers. The user types a card image per line according to a format appropriate for the programming language. (The supervisor collects these card images at the end of the user's private file tape.) When in the automatic input mode, the manual mode may be entered by giving an initial carriage return and typing the appropriate line number followed by | and line for as many lines as desired. To reenter the automatic mode, an initial carriage return is given.

The manual mode allows the user to overwrite previous lines and to insert lines. (cf. File Command.)

5. edit | α | β

α = title of file

β = class of file

The user is set in the automatic input mode with the designated file treated as initial input lines. The same conventions apply as to the input command.

6. file | α | β

α = title to be given to file

β = class of language used during input

The created file will consist of the numbered input lines (i.e. those at the end of the user's private file tape) in sequence; in the case of duplicate line numbers, the last version will be used. The line numbers will be written as sequence numbers in the corresponding card images of the file.

For convenience the following editing conventions apply to input lines:

a. an underline signifies the deletion of the previous characters of the line.

b. a backspace signifies the deletion of the previous character in the field.

The following formats apply:

a. FAP: symbol, tab, operation, tab, variable field and comment.

b. MAD, MADTRAN, FORTRAN: statement label, tab, statement. To place a character in the continuation column: statement label, tab, backspace, character, statement.

c. DATA: cols. 1-72.

7. fap | α

Causes the file designated as α, fap to be translated by the FAP translator (assembler). Files α, symtb and α,bss are added to the user's private file tape giving the symbol table and the relocatable binary BSS form of the file.

8. mad | α

Causes file α,mad to be translated by the MAD translator (compiler). File α,bss is created.

9. madtrn | α

Causes file α,madtrn (i.e. a pseudo-Fortran language file) to be edited into an equivalent file α,mad (added to the user's file) and translation occurs as if the command mad|α had been given.

10. load | α_1 | α_2|...|α_n

Causes the consecutive loading of files α_i,bss (i=1,2,...,n). An exception occurs if α_i= (libe), in which case file α_{i+1},bss is searched as a library file for all subprograms still missing. (There can be further library files.)

11. use | α_1 | α_2 |...| α_n

This command is used whenever a load or previous use command notifies the user of an incomplete set of subprograms. Same α_i conventions as for load.

12. start | α | β

Starts the program setup by the load and use commands (or a dormant program) after first positioning the user private file tape in front of the title card for file α,β. (If β is not given, a class of data is assumed; if both α and β are not given, no tape movement occurs and the program is started.)

13. pm | α

α = "lights", "stomap", or the usual format of the standard Center post-mortem (F2PM) request: subprogram name | loc_1 | loc_2 | mode | direction where mode and direction are optional.

Produces post-mortem of user's dormant program according to request specified by α. (E.g. matrix | 5 | 209 | flo | rev will cause to be printed on the user's typewriter the contents of subprogram "matrix" from relative locations 5 to 209 in floating point form and in reverse sequence.)

14. skippm

Used if a pm command is "quit" during output and the previous program interruption is to be restarted.

15. listf

Types out list of all file titles on user's private file tape.

16. printf | α | β | γ

Types out file α,β starting at line number γ. If γ is omitted, the initial line is assumed. Whenever the user's output buffer fills, the command program goes into an I/O wait status allowing other users to time-share until the buffer needs refilling.

17. xdump | α | β

Creates file α,β (if β omitted, xdump assumed) on user's private file tape consisting of the complete state of the user's last dormant program.

18. xdump | α | β

Inverse of xdump command in that it resets file α,β as the user's program, starting it where it last left off.

Although experience with the system to date is quite limited, first indications are that programmers would readily use such a system if it were generally available It is useful to ask, now that there is some operating experience with the 7090 system, what observations can be made. An immediate comment is that once a user gets accustomed to computer response, delays of even a fraction of a minute are exasperatingly long, an effect analogous to conversing with a slow-speaking person. Similarly, the requirement that a complete typewritten line rather than each character be the minimum unit of man-computer communication is an inhibiting factor in the sense that a press-to-talk radio-telephone conversation is more stilted than that of an ordinary telephone. Since maintaining a rapid computer response on a character by character basis requires at least a vestigial response program in core memory at all times, the straightforward solution within the present system is to have more core memory available. At the very least, an extra bank of memory for the time-sharing supervisor would ease compatibility problems with programs already written for 32,000 word 7090's.

For reasons of expediency, the weakest portions of the present system are the conventions for input, editing of user files, and the degree of rapid interaction and intimacy possible while debugging. Since to a large extent these areas involve the taste, habits, and psychology of the users, it is felt that proper solutions will require considerable experimentation and pragmatic evaluation; it is also clear that these areas cannot be treated in the abstract for the programming languages used will influence greatly the appropriate techniques. A greater use of symbolic referencing for locations, program names and variables is certainly desired; symbolic post-mortem programs, trace programs, and before-and-after differential dump programs should play useful roles in the debugging procedures.

In the design of the present system, great care went into making each user independent of the other users. However, it would be a useful extension of the system if this were not always the case. In particular, when several consoles are used in a computer controlled group such as in management or war games, in group behavior studies, or possibly in teaching machines, it would be desirable to have all the consoles communicating with a single program.

Another area for further improvement within the present system is that of file maintenance, since the presently used tape units are a hindrance to the easy deletion of user program files. Disc units will be of help in this area as well as with the problem of consolidating and scheduling large-scale central input-output generated by the many console users.

Finally, it is felt that it would be desirable to have the distinction between the foreground and background systems eliminated. The present-day computer operator would assume the role of a stand-in for the background users, using an operator console much like the other user consoles in the system, mounting and demounting magnetic tapes as requested by the supervisor, receiving instructions to read card decks into the central disc unit, etc. Similarly the foreground user, when satisfied with his program, would by means of his console and the supervisor program enter his program into the queue of production background work to be performed. With these procedures implemented the distinction of whether one is time-sharing or not would vanish and the computer user would be free to choose in an interchangable way that mode of operation which he found more suitable at a particular time.

A Multi-Level Scheduling Algorithm

Regardless of whether one has a million word core memory or a 32,000 word memory as currently exists on the 7090, one is inevitably faced with the problem of system saturation where the total size of active user programs exceeds that of the high-speed memory or there are too many active user programs to maintain an adequate response at each user console. These conditions can easily arise with even a few users if some of the user programs are excessive in size or in time requirements. The predicament can be alleviated if it is assumed that a good design for the system is to have a saturation procedure which gives graceful degradation of the response time and effective real-time computation speed of the large and long-running users.

To show the general problem, Figure 1 qualitatively gives the user service as a function of n, the number of active users. This service parameter might be either of the two key factors: computer response time or n times the real-time computation speed. In either case there is some critical number of active users, N, representing the effective user capacity, which causes saturation. If the strategy near saturation is to execute the simple round-robin of all users, then there is an abrupt collapse of service due to the sudden onset of the large amount of time required to swap programs in-and-out of the secondary memory such as a disc or drum unit. Of course, Figure 1 is quite qualitative since it depends critically on the spectrum of user program sizes as well as the spectrum of user operating times.

To illustrate the strategy that can be employed to improve the saturation performance of a time-sharing system, a multi-level scheduling algorithm is presented. This algorithm also can be analyzed to give broad bounds on the system performance.

The basis of the multi-level scheduling algorithm is to assign each user program as it enters the system to be run (or completes a response to a user) to an ℓth level priority queue. Programs are initially entered into a level ℓ_o, corresponding to their size such that

$$\ell_o = \left[\log_2\left(\left[\frac{w_p}{w_q}\right] + 1\right)\right] \qquad (1)$$

where w_p is the number of words in the program, w_q is the number of words which can be transmitted in <u>and</u> out of the high-speed memory from the secondary memory in the time of one quantum, q, and the bracket indicates "the integral part of". Ordinarily the time of a quantum, being the basic time unit, should be as small as possible without excessive overhead losses when the supervisor switches from one program in high-speed memory to another. The process starts with the time-sharing supervisor operating the program at the head of the lowest level occupied queue, ℓ, for up to 2^ℓ quanta of time and then if the program is not completed (i.e. has not made a response to the user) placing it at the end of the ℓ+1 level queue. If there are no programs entering the system at levels lower than ℓ, this process proceeds until the queue at level ℓ is exhausted; the process is then iteratively begun again at level ℓ+1, where now each program is run for $2^{\ell+1}$ quanta of time. If during the execution of the 2^ℓ quanta of a program at level ℓ, a lower level, ℓ', becomes occupied, the current user is replaced at the head of the ℓth queue and the process is reinitiated at level ℓ'.

Similarly, if a program of size w_p at level ℓ, during operation requests a change in memory size from the time-sharing supervisor, then the enlarged (or reduced) version of the program should be placed at the end of the ℓ'' queue where

$$\ell'' = \ell + \left[\log_2\left(\left[\frac{w_p''}{w_p}\right] + 1\right)\right] . \qquad (2)$$

Again the process is re-initiated with the head-of-the-queue user at the lowest occupied level of ℓ'.

Several important conclusions can be drawn from the above algorithm which allow the performance of the system to be bounded.

<u>Computational Efficiency</u>

1. Because a program is always operated for a time greater than or equal to the swap time (i.e. the time required to move the program in and out of secondary memory), it follows that the computational efficiency never falls below one-half. (Clearly, this fraction is adjustable in the formula for the initial level, ℓ_o.) An alternative way of viewing this bound is to say that the real-time computing speed available to one out of n active users is no worse than if there were 2n active users all of whose programs were in the high-speed memory.

<u>Response Time</u>

2. If the maximum number of active users is N, then an individual user of a given program size can be guaranteed a response time,

$$t_r \leqslant 2Nq\left(\left[\frac{w_p}{w_q}\right] + 1\right) \qquad (3)$$

since the worst case occurs when all competing user programs are at the same level. Conversely, if t_r is a guaranteed response of arbitrary value and the largest size of program is assumed, then the maximum permissible number of active users is bounded.

<u>Long Runs</u>

3. The relative swap time on long runs can be made vanishingly small. This conclusion follows since the longer a program is run, the higher the level number it cascades to with a correspondingly smaller relative swap time. It is an important feature of the algorithm that long runs must in effect prove they are long so that programs which have an unexpected demise are detected quickly. In order that there be a finite number of levels, a maximum level number, L, can be established such that the asymptotic swap overhead is some arbitrarily small percentage, p:

Note: formula (2) is incorrect. To correct, remove "+ 1" and remove inner brackets around ratio of program sizes.

$$L = \left[\log_2 \left(\left[\frac{w_{pmax}}{pw_q} \right] + 1 \right) \right] \quad (4)$$

where w_{pmax} is the size of the largest possible program.

<u>Multi-level vs. Single-level Response Times</u>

4. The response time for programs of equal size, entering the system at the same time, and being run for multiple quanta, is no worse than approximately twice the response-time occurring in a single quanta round-robin procedure. If there are n equal sized programs started in a queue at level ℓ, then the worst case is that of the end-of-the-queue program which is ready to respond at the very first quantum run at the ℓ+j level. Using the multi-level algorithm, the total delay for the end-of-the-queue program is by virtue of the geometric series of quanta:

$$T_m \sim q2^{\ell} \left\{ n(2^j-1) + (n-1)\ 2^j. \right\} \quad (5)$$

Since the end-of-the-queue user has computed for a time of $2^{\ell}(2^j-1)$ quanta, the equivalent single-level round-robin delay before a response is:

$$T_s \sim q2^{\ell} \left\{ n(2^j-1) \right\}. \quad (6)$$

Hence

$$\frac{T_m}{T_s} \sim 1 + \left(\frac{n-1}{n} \right) \left(\frac{2^j}{2^j-1} \right) \sim 2 \quad (7)$$

and the assertion is shown. It should be noted that the above conditions, where program swap times are omitted, which are pertinent when all programs remain in high-speed memory, are the least favorable for the multi-level algorithm; if swap times are included in the above analysis, the ratio of T_m / T_s can only become smaller and may become much less than unity. By a similar analysis it is easy to show that even in the unfavorable case where there are no program swaps, head-of-the-queue programs that terminate just as the $2^{\ell+j}$ quanta are completed receive under the multi-level algorithm a response which is twice as <u>fast</u> as that under the single-level round-robin (i.e. $T_m / T_s = 1/2$).

<u>Highest Serviced Level</u>

5. In the multi-level algorithm the level classification procedure for programs is entirely automatic, depending on performance and program size rather than on the declarations (or hopes) of each user. As a user taxes the system, the degradation of service occurs progressively starting with the higher level users of either large or long-running programs; however, at some level no user programs may be run because of too many active users at lower levels. To determine a bound on this cut-off point we consider N active users at level ℓ each running 2^{ℓ} quanta, terminating, and reentering the system again at level ℓ at a user response time, t_u, later. If there is to be no service at level ℓ+1, then the computing time, $Nq2^{\ell}$, must be greater than or equal to t_u. Thus the guaranteed active levels, ℓ_a, are given by the relation:

$$\ell_a \lesssim \left[\log_2 \left(\frac{t_u}{Nq} \right) \right] \quad (8)$$

In the limit, t_u could be as small as a minimum user reaction time ($\sim$.2 sec.), but the expected value would be several orders of magnitude greater as a result of the statistics of a large number of users.

The multi-level algorithm as formulated above makes no explicit consideration of the seek or latency time required before transmission of programs to and from disc or drum units when they are used as the secondary memory, (although formally the factor w_q could contain an average figure for these times). One simple modification to the algorithm which usually avoids wasting the seek or latency time is to continue to operate the last user program for as many quanta as are required to ready the swap of the new user with the least priority user; since ordinarily only the higher level number programs would be forced out into the secondary memory, the extended quanta of operation of the old user while seeking the new user should be but a minor distortion of the basic algorithm.

Further complexities are possible when the hardware is appropriate. In computers with input-output channels and low transmission rates to and from secondary memory, it is possible to overlap the reading and writing of the new and old users in and out of high-speed memory while operating the current user. The effect is equivalent to using a drum giving 100 % multiplexor usage but there are two liabilities, namely, no individual user can utilize all the available user memory space and the look-ahead procedure breaks down whenever an unanticipated scheduling change occurs (e.g. a program terminates or a higher-priority user program is initiated).

Complexity is also possible in storage allocation but certainly an elementary procedure and a desirable one with a low-transmission rate secondary memory is to consolidate in a single block all high-priority user programs whenever sufficient fragmentary unused memory space is available to read in a new user program. Such a procedure is indicated in the flow diagram of the multi-level scheduling algorithm which is given as Figure 2.

It should also be noted that Figure 2 only accounts for the scheduling of programs in a working status and still does not take into account the storage allocation of programs which are in a dormant (or input-output wait status). One systematic method of handling this case is to modify the scheduling algorithm so that programs which become dormant at level ℓ are entered into the queue at level $\ell+1$. The scheduling algorithm proceeds as before with the dormant programs continuing to cascade but not operating when they reached the head of a queue. Whenever a program must be removed from high-speed memory, a program is selected from the end-of-the-queue of the highest occupied level number.

Finally, it is illuminating to apply the multi-level scheduling algorithm bounds to the contemporary IBM 7090. The following approximate values are obtained:

q = 16 m.s. (based on 1% switching overhead)

w_q = 120 words (based on one IBM 1301 model 2 disc unit without seek or latency times included)

$t_r \leqslant 8Nf_{sec.}$ (based on programs of (32k)f words)

$\ell_a \leqslant \log_2 (1000/N)$ (based on t_u = 16 sec.)

$\ell_o \leqslant 8$ (based on a maximum program size of 32K words)

Using the arbitrary criteria that programs up to the maximum size of 32,000 words should always get some service, which is to say that max ℓ_a = max ℓ_o , we deduce as a conservative estimate that N can be 4 and that at worst the response time for a trivial reply will be 32 seconds.

The small value of N arrived at is a direct consequence of the small value of w_q that results from the slow disc word transmission rate. This rate is only 3.3% of the maximum core memory multiplexor rate. It is of interest that using high-capacity high-speed drums of current design such as in the Sage System of in the IBM Sabre System it would be possible to attain nearly 100% multiplexor utilization and thus multiply w_q by a factor of 30. It immediately follows that user response times equivalent to those given above with the disc unit would be given to 30 times as many persons or to 120 users; the total computational capacity, however, would not change.

In any case, considerable caution should be used with capacity and computer response time estimates since they are critically dependent upon the distribution functions for the user response time, t_u, and the user program size, w_p, and the computational capacity requested by each user. Past experience using conventional programming systems is of little assistance because these distribution functions will depend very strongly upon the programming systems made available to the time-sharing users as well as upon the user habit patterns which will gradually evolve.

Conclusions

In conclusion, it is clear that contemporary computers and hardware are sufficient to allow moderate performance time-sharing for a limited number of users. There are several problems which can be solved by careful hardware design, but there are also a large number of intricate system programs which must be written before one has an adequate time-sharing system. An important aspect of any future time-shared computer is that until the system programming is completed, especially the critical time-sharing supervisor, the computer is completely worthless. Thus, it is essential for future system design and implementation that all aspects of time-sharing system problems be explored and understood in prototype form on present computers so that major advances in computer organization and usage can be made.

Acknowledgements

The authors wish to thank Bernard Galler, Robert Graham and Bruce Arden, of the University of Michigan, for making the MAD compiler available and for their advice with regard to its adaption into the present time-sharing system. The version of the Madtran Fortran-to-Mad editor program was generously supplied by Robert Rosin of the University of Michigan. Of the MIT Computation Center staff, Robert Creasy was of assistance in the evaluation of time-sharing performance, Lynda Korn is to be credited for her contributions to the pm and madtran commands, and Evelyn Dow for her work on the fap command.

References

1. Strachey, C., "Time Sharing in Large Fast Computers," Proceedings of the International Conference on Information Processing, UNESCO (June, 1959), Paper B.2.19.

2. Licklider, J. C. R., "Man-Computer Symbiosis," IRE Transactions on Human Factors in Electronics, HFE-1, No. 1 (March, 1960), 4-11.

3. Brown, G., Licklider, J. C. R., McCarthy, J., and Perlis, A., lectures given spring, 1961, Management and the Computer of the Future, (to be published by the M.I.T. Press, March, 1962).

4. Corbató, F. J., "An Experimental Time-Sharing System," Proceedings of the IBM University Director's Conference, July, 1961 (to be published).

5. Schmitt, W. F., Tonik, A. B., "Sympathetically Programmed Computers," Proceedings of the International Conference on Information Processing, UNESCO, (June, 1959) Paper B.2.18.

6. Codd, E. F., "Multiprogram Scheduling," Communications of the ACM, 3, 6 (June, 1960), 347-350.

7. Heller, J., "Sequencing Aspects of Multiprogramming," Journal of the ACM, 8, 3 (July, 1961), 426-439.

8. Leeds, H. D., Weinberg, G. M., "Multiprogramming," Computer Programming Fundamentals, 356-359, McGraw-Hill (1961).

9. Teager, H. M., "Real-Time Time-Shared Computer Project," Communications of the ACM, 5, 1 (January, 1962) Research Summaries, 62.

10. Teager, H. M., McCarthy, J., "Time-Shared Program Testing," paper delivered at the 14th National Meeting of the ACM (not published).

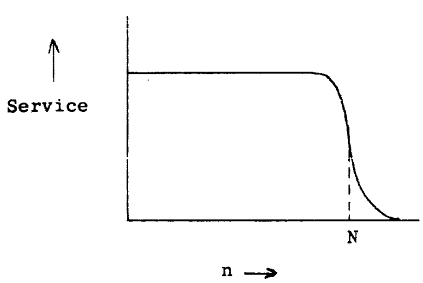

Figure 1. Service vs. Number of Active Users

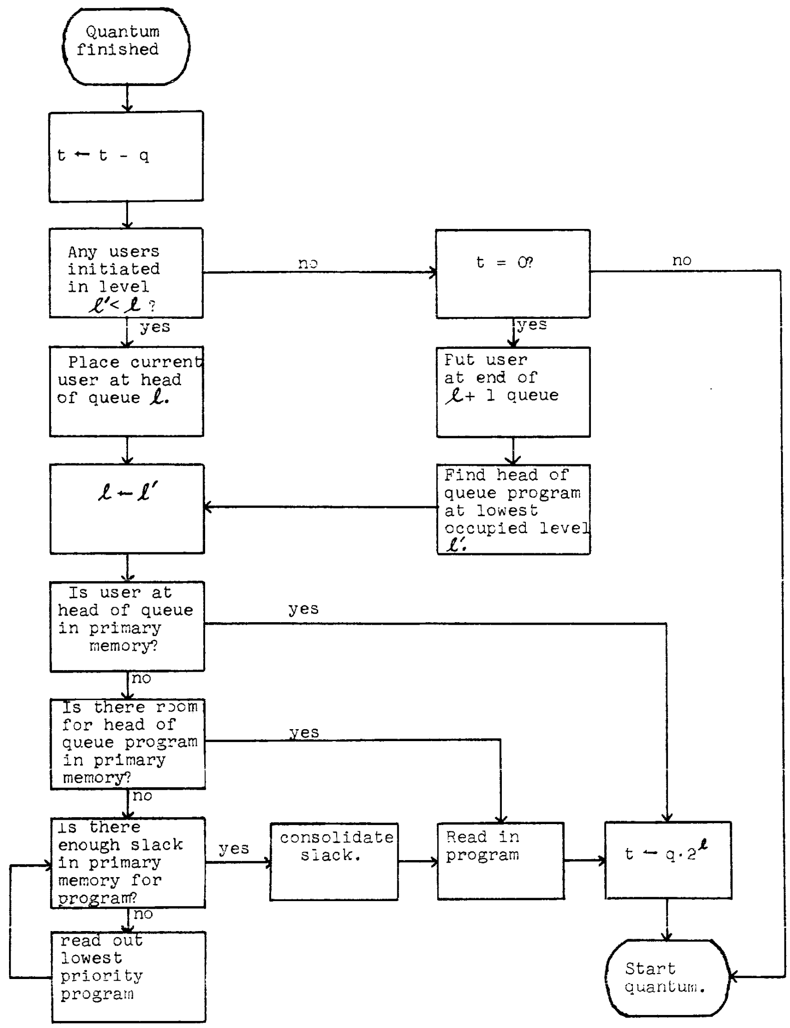

Figure 2. Flow Chart of Multi-Level Scheduling Algorithm