D. D. Chamberlin
H. P. Schlaeppi
I. Wladawsky

Experimental Study of Deadline Scheduling for Interactive Systems

Abstract: This paper outlines a resource allocation strategy called deadline scheduling, which is intended for use in interactive systems. Experiments are reported in which simplified versions of deadline scheduling and two time slicing strategies are modeled and compared under identical conditions. Results suggest that deadline scheduling, primarily by reducing paging overhead, provides faster response and supports more interactive users concurrently than do the two time slicing methods.

Introduction

Performance of interactive computer systems is judged in accordance with several different standards, such as transaction throughput and resource utilization. However, these criteria usually rank behind subjective judgments of terminal response time by users of the system. Response times are usually judged relative to the users' estimate of the productive work done by the system in processing each transaction. For example, a user may expect a very fast response to a trivial editing command, but be content to wait several minutes for compilation of a large program. Users' estimates of the complexity of their transactions do not always correspond to the actual load placed by these transactions on the resources of the system. Nevertheless, a system that is consistently slow in responding to seemingly simple commands will frustrate its users and will be judged accordingly.

The performance of an interactive system depends to a large extent on the scheduling algorithm that controls the allocation of system resources among users' transactions. Much work has gone into devising and refining these algorithms [1–8]. A number of distinct approaches have evolved, such as multilevel time slicing [1] and policy-driven scheduling [2]. Generally, the idea underlying all of them is to distribute the system's resources in some sense equitably over the set of transactions that are currently in process. Usually, this is achieved by, in effect, decreasing the "eligibility" of a transaction to the extent that it is consuming resources, so that at some point in time it is forced to yield its resources to a competing transaction (as in "time slice end").

In general, current scheduling strategies do not make explicit use of the user's expectation with regard to a transaction. Rather, they accumulate information about each transaction by observing the transaction as it is being executed (for example, a long transaction may be treated the same as a short one until some number of time slices have elapsed.) This principle has the advantage that no *a priori* information is required about the transactions. It has the disadvantages that the measurement process (e.g., time slicing) results in nonproductive overhead (e.g., repeated page fetches), and that inappropriate allocations may be made during the initial period before the system has gathered sufficient information about a new transaction. For example, a long transaction, which should be given low priority, is treated with high priority until its nature is recognized.

An alternative approach to scheduling, called deadline scheduling, has been proposed [9] and is outlined in this paper. Deadline scheduling explicitly uses the reasonably expected response time of each transaction as a scheduling parameter called a *deadline*. System resources are allocated according to a priority ordering among transactions, where priority is based on the relative closeness of each transaction to its deadline. It was expected that deadline scheduling would result in the following advantages over a schedular based on time slicing:

1. A reduction in nonintrinsic overhead due to CPU-switching, page swapping, etc.,
2. A more favorable distribution of response times, and

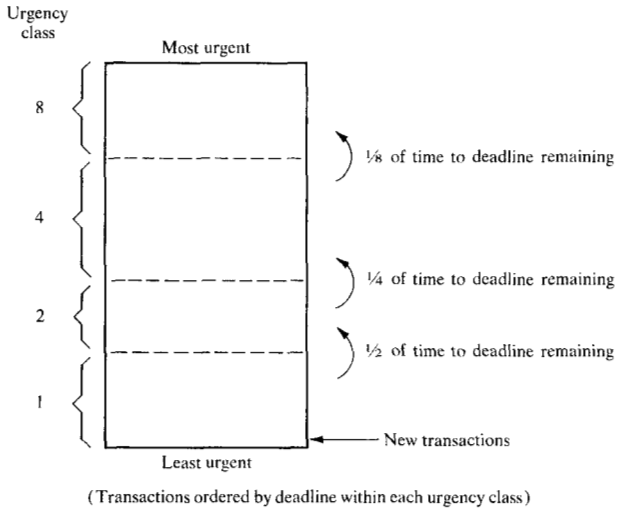

Figure 1 Scheduling stack for deadline scheduling.

3. The ability to support a larger number of concurrently active users with a given response quality.

In this paper we report on a modeling experiment designed to investigate these contentions. We first describe simplified models of a deadline scheduler and of two schedulers based on time slicing. We then outline the general structure of the experiment and describe the load to which the schedulers were subjected. Finally we report and discuss the experimental results.

Deadline scheduling

Deadline scheduling assumes that each transaction, when it arrives in the system, has a completion deadline. This approach creates a new problem, i.e., translating *a priori* information about each transaction into an appropriate deadline. Since this problem is not the subject of this paper, it suffices to observe that the nature of commands (e.g., EDIT, EXECUTE) and their parameters (e.g., file size) are generally adequate for determining a first approximation to a user-acceptable deadline. Deadlines could thus be assigned by higher-level subsystems (such as editors, compilers, and command interpreters) according to the characteristics of the transaction. (This would of course incur some additional overhead, which is not reflected in our model.) The original estimate could be refined dynamically if necessary on the basis of the measured behavior of the transaction.

In our model, transactions are sorted into a number of discrete categories, each with its own deadline. These categories are described in greater detail later. In the simplified version of deadline scheduling used in the model, each transaction is assigned an initial *urgency* of 1 when it arrives in the system. Its urgency is then doubled whenever half of the remaining time to its deadline has elapsed, until the urgency reaches a *ceiling* (8 for the runs described here). Thus, as illustrated in Fig. 1, all transactions in the system are dynamically ordered in a scheduling stack according to their urgency, and within a given urgency class according to their deadline.

At each instant, a working page set size [9] is associated with each transaction. (The exact meaning of this term and the method by which working set size is generated are described later.) Pages that spontaneously become available (for example, by termination of a transaction) are allocated to the most urgent transaction that has fewer allocated pages than its working set size. CPU's that spontaneously become available are given to the most urgent transaction that has its full set of allocated pages but lacks a CPU. (The strategy is capable of handling multiple CPU's.) Partial page allocations, i.e., allocations of fewer pages than a transaction's full working set size, may be stolen and given at any time to a more urgent transaction that requests more pages. Full page allocations, however, are never stolen unless some transaction's urgency reaches the ceiling.

When the urgency of any transaction reaches the ceiling, the system reconsiders the allocation of all resources. The system scans the transaction stack (see Fig. 1) from the top, ensuring that CPU's and pages are allocated to the topmost transactions, stealing resources from less urgent transactions as necessary. To steal resources, the system scans the stack from the bottom. It is expected that resources will be stolen in this way only rarely, since the urgencies of the various transactions tend to remain in the same relative order as all the transactions progress toward their respective deadlines (the main exception is due to newly arriving transactions of relatively early deadlines). To the extent that this is true, the rate of forced page fetches and task switches remains relatively low as the system load approaches capacity, and "thrashing" is avoided.

Time slicing strategies

For the simplified time slicing strategies to be used in the model, we define a "multiprogramming ring," which at any given instant contains some subset of the transactions known to the system. Only the transactions in the multiprogramming ring are considered candidates for allocation of pages or CPU's. Transactions not in the multiprogramming ring are said to be *dormant*.

The number of transactions in the multiprogramming ring is governed by the number of pages in memory. The "most eligible" dormant transaction is promoted into the multiprogramming ring when there are enough unallocated pages in memory to accommodate its current working set, plus a small additional reserve of unused pages. When a transaction is placed in the multiprogramming

ring, it is immediately allocated enough pages to hold its working set. CPU's are allocated among transactions in the multiprogramming ring on a round-robin basis.

A transaction remains in the multiprogramming ring until it has used a fixed amount of CPU time (called a time slice) or until it requests allocation of more pages than are currently available. At this time the transaction is made dormant and its pages are made available for other transactions. The reserve of unused pages enables the system to satisfy new requests for pages and thus prevents frequent demotions of running transactions to the dormant stage. For the experiments described in this paper, the reserve has been set to ten pages.

The two time slicing strategies differ only in their definition of the "most eligible" dormant transaction. In one-level time slicing, the most eligible dormant transaction is the transaction that has been dormant for the longest time. In two-level time slicing, the most eligible transaction is the longest-dormant transaction that has not yet completed a time slice; if none, then the longest-dormant transaction that has completed one or more time slices.

Figures 2 and 3 illustrate one- and two-level time slicing, respectively.

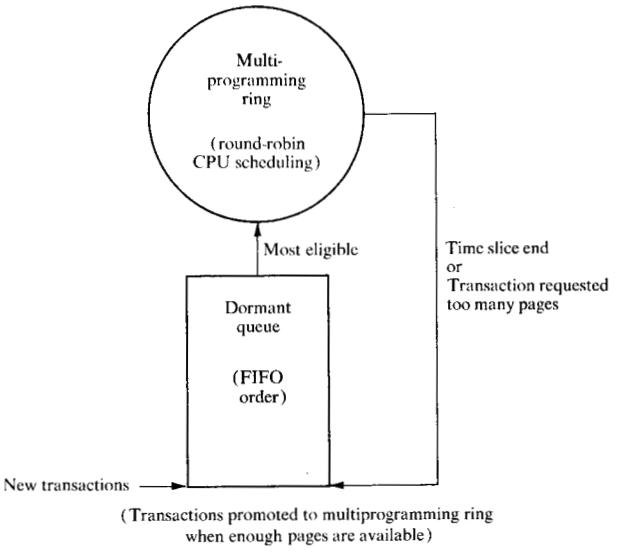

Figure 2 One-level time slicing.

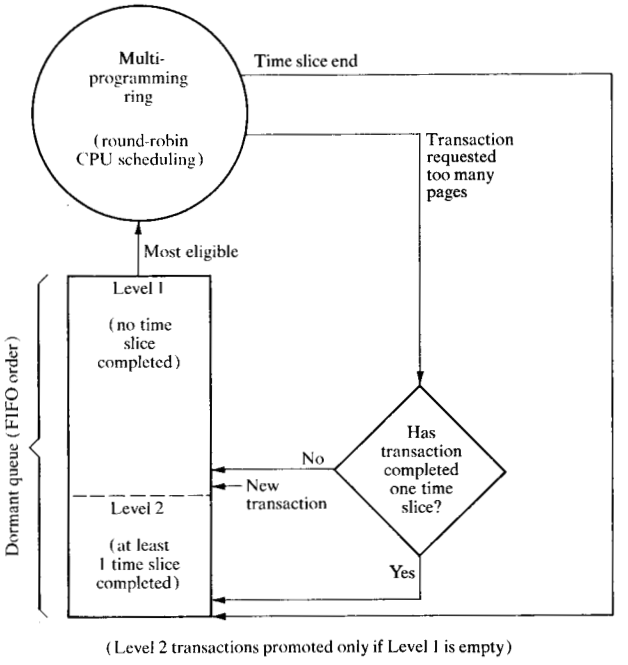

Figure 3 Two-level time slicing.

The model

In our model, discrete event techniques are used to simulate the allocation of processing time and main memory to a number of contending transactions, neglecting contention for other resources. This is a rather simplified view of an operating system. The model is nevertheless believed to be meaningful because the simplifications have comparable effects on all strategies modeled, and because the interacting effects of contention for more than one class of resources (processors and memory space) are included in the model.

The model takes into account the delay incurred in fetching the contents of a newly referred to page. No attempt is made to keep track of the identity of page contents. The user of the model specifies, as parameters of each run, the numbers of CPU's and pages available in the system, the page fetch time, the characteristics of the system load, and which of the three scheduling strategies is to be used.

The model, which was coded in PL/1, consists of three parts:

1. A transaction stream generator, which, at the beginning of a run, generates the stream of transactions used to load the system during the run.
2. A calendar-driven simulator. At each instant of simulated time, the calendar contains an ordered list of all future events known to the model. The basic cycle, which is executed repeatedly, is as follows:
   a. Update simulated time to the nearest future event.
   b. Simulate this event. (This may generate additional future events that are placed on the calendar in their proper chronological order.)
3. An output analysis routine, which gathers and prints data on the behavior of the system.

The load

Before a simulation run begins, a set of transactions is generated to serve as the system load. Each transaction is described by the time of its arrival in the system and by its processing and storage needs. The processing demand of a transaction is simply given by its total execution time. The most detailed description possible of the storage demands of a given transaction is a complete list of its page references. For the purpose of this model, however, we have chosen to characterize storage demands by assigning to each transaction a "working page set size", which fluctuates as the transaction is executed. We assume that execution of a transaction can continue only if it is allocated a number of pages at least equal to its working set size.

In this model the working set includes all the pages referred to by a transaction in a certain time period (until the next working set size change). If the working set decreases, pages are released to the system; if it increases requests for additional pages are issued.

Various algorithms have been proposed for the approximate measurement of working set size as a dynamic parameter [9]. Our experiment replaces such an algorithm by an *a priori* generated distribution. The working set size of each transaction is generated, as a function of elapsed execution time of the transaction, by the load generator, and is made available for use by the various scheduling strategies. Thus, the profile of each transaction is given by its arrival time and a list of "events" that occur after the transaction has accumulated certain amounts of CPU time. These events represent changes in the working set size. For each transaction the working set size fluctuates about a mean. The final event of each transaction is its termination.

A typical time-sharing system must be expected to handle several types of transactions, each with its own unique set of characteristics; for example, it might simultaneously run interactive terminal transactions and background compute-bound transactions. Our model reflects this by allowing the user to specify several "categories" of transactions. The total load on the system is a combination of transactions generated from all the categories.

The arrival times and resource requirements of the transactions, and the timing of the events within each transaction, are randomly generated according to the distributions to be described. For each category of transactions, the model user specifies:

1. Mean interarrival time. (The model distributes actual transaction arrivals randomly throughout the simulated run according to a uniform distribution.)
2. The mean and standard deviation of CPU time used before termination. (The model uses a Gaussian distribution for assigning CPU time requirements, modified by taking the absolute value of the generated time. A Gaussian distribution was chosen because it was felt that separate parameters were needed for controlling the mean and the standard deviation).
3. The deadline for transactions in the category, expressed as elapsed real time after arrival in the system. (The model assumes identical deadlines for all transactions in a category.)
4. The mean working page set size among all transactions in the category. (The mean working set for a particular transaction is chosen from a Poisson distribution; the instantaneous working set of a particular transaction then varies randomly as a function of execution time about the mean for that transaction, again using a Poisson distribution.)

Experimental results

Our experimental results are based on a simulated system having two CPU's and one megabyte of main storage. We assume 4K bytes per page and that 150 pages are occupied by the resident system, leaving 100 pages for user transactions. The simulated page fetch time is set at 20 ms.

For the first part of our experiment we wish to measure the relative capacities of the three strategies to support interactive terminal users in the presence of a background load. Accordingly, we set the load generator to produce a category of "background" transactions. The following characteristics were chosen for this category:

Mean arrival rate:	3 per min
Mean CPU usage:	16 s
Mean working set size:	50 pages
Deadline:	180 s
CPU load factor:	40%

A CPU load factor of 40% denotes that the background transactions alone would consume 40% of the capacity of the system's two CPU's, neglecting idleness due to contention for pages.

In addition to the background transaction category, we set the load generator to produce a category of "terminal" transactions having the following characteristics:

Mean CPU usage:	0.8 s
Mean working set size:	20 pages
Deadline:	4 s

We vary the mean arrival rate of terminal transactions from one experiment to another in order to simulate various numbers of terminal users. Statistics on existing interactive systems indicate that the average terminal user generates a transaction about every 30 to 60 seconds. (See Appendix.) Using the 30-second figure, Table 1 shows the approximate numbers of terminal users that correspond to the various arrival rates used in the

experiments, and the CPU load factor for each. We see that a mean arrival rate of 90 interactive transactions per minute, in addition to the background load, produces a total CPU load factor of 100 percent. Since any real resource allocation strategy will result in some CPU idleness while pages are being fetched, we do not expect any of the strategies modeled to successfully handle 90 interactive transactions per minute.

It might be noted here that the inhomogeneous nature of the load is such that a simple first-come, first-served scheduler without time slicing would be quite unsuitable. In such a scheduling discipline two background transactions would seize the two CPU's and hold them until completion, during which time a great many terminal transactions would arrive in the queue. The result would be unacceptably long response times for terminal transactions.

In order to fairly compare the load-handling ability of deadline and time slicing schedulers, we have optimized the time slicing schedulers by selecting for each the time slice length capable of handling the largest load. For this purpose, a series of runs were made to find the maximum arrival rate of terminal transactions (in addition to the background load) that could be supported by the two time slicing strategies for various time slice lengths. In each run, a stream of transaction arrivals was generated statistically as described above for the first 100 simulated seconds of the run. This input transaction stream was then duplicated in each successive 100 simulated seconds until the end of the run. A record was kept of queue length (number of transactions that have arrived but not yet terminated) vs. time for at least 1200 simulated seconds of each run. The system was judged to support the load if the queue length became stable and periodic with period equal to some multiple of 100 seconds (the input period). If the queue grew without bound as a linear function of time, the system was judged not to support the load.

The maximum transaction arrival rate supported by the three scheduling strategies is plotted in Fig. 4. The deadline scheduler has no parameter that can be "tuned" in analogy to the time slice length, since deadlines are properties of the transactions rather than of the strategy. The regions of uncertainty marked on the curves of Fig. 4 denote experiments for which the system queue length neither became periodic nor grew linearly. These uncertainties are due to the stochastic nature of the experiments. We see from the plot that deadline scheduling supports the highest rate of interactive transactions under the given conditions, followed by two-level time slicing and one-level time slicing. We also note that the load-handling ability of the time slicing strategies drops sharply as the time slice approaches (from above) the average duration of a terminal transaction. This phenomenon is probably due to the lost time involved in refetching the pages belonging to a terminal transaction if that transaction requires more than one time slice to complete.

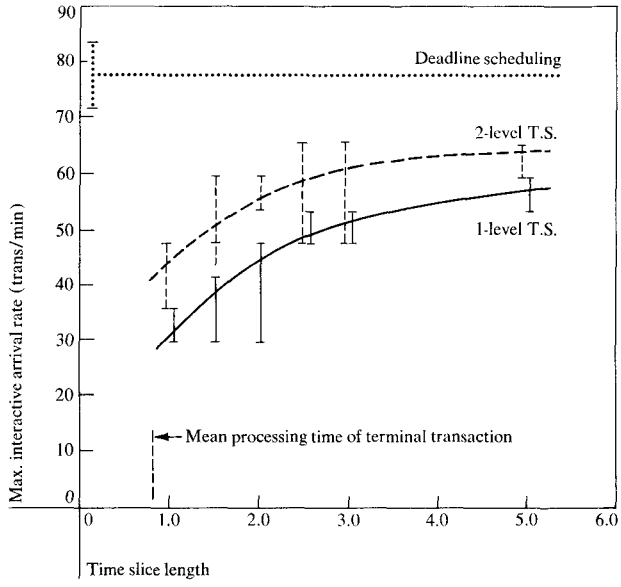

Figure 4 Maximum load supported by three scheduling strategies.

Table 1 Number of terminal users corresponding to arrival rates used in experiments.

Mean interactive arrival rate (trans./min)	*Approx. no. of terminal users*	*CPU load factor for terminal transactions (percent)*
30	15	20
36	18	24
42	21	28
48	24	32
54	27	36
60	30	40
66	33	44
72	36	48
78	39	52
84	42	56
90	45	60

For the second part of our experiment, we wish to compare the responsiveness of deadline scheduling to that of the other two strategies under the same load. We use the deadlines as a standard of performance for all three strategies, although only the deadline scheduler uses them explicitly as scheduling parameters. We define a "response ratio" as the turnaround time of a transaction divided by its deadline. All transactions with a response ratio greater than one are "late."

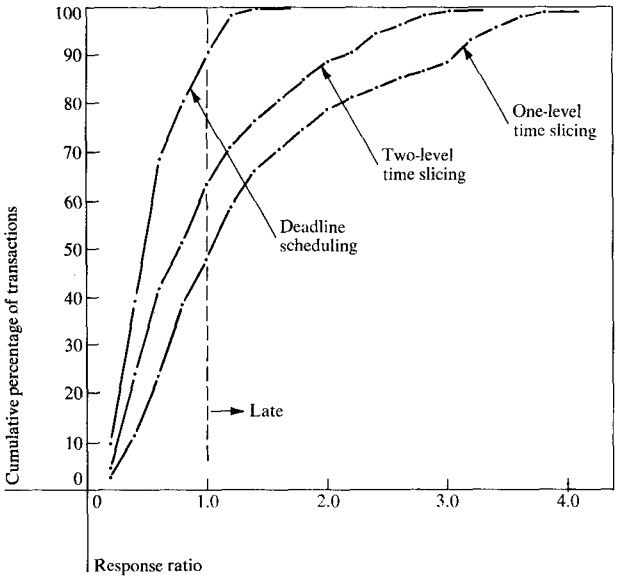

Figure 5 Response ratio distributions for three scheduling strategies.

We will compare the responsiveness of the three strategies, loaded with 48 interactive transactions per minute plus background, which was the largest load that all three strategies were able to support. Once again, in order to be fair to the time slicing strategies, we determine the time slice length that produces the most favorable results. For the one-level and two-level time slicing strategies, a series of runs were made with the above load. The runs used various time slice lengths, and each run had a duration of 20 simulated minutes. The average response ratio of all terminal transactions in each run is shown in Table 2. We see that there is no simple relationship between time slice length and average response ratio for either of the time slicing strategies. Nevertheless, we selected the time slice length that produced the most favorable average response ratio for each strategy (3.5 s for one-level, 2.0 s for two-level.)

Next we investigated the reponse-time distribution of each time slicing strategy more fully, using its most favorable time slice length as determined above, and compared it with that of deadline scheduling. A run of 30 simulated minutes was made, with a load of 48 interactive transactions per minute plus background, for each of the strategies. For each run, a plot was made of the distribution of response ratios among all transactions in the run (see Fig. 5.) We see that the most favorable distribution of response ratios (greatest percentage of transactions "on time" or "early") is produced by deadline scheduling, followed by two-level time slicing and one-level time slicing.

For the same runs that produced the results in Fig. 5, a count was kept of the total number of page fetches and

Table 2 Average response ratio for interactive transactions (load = 48 interactive trans./min plus background).

Time slice length (s)	*One-level time slicing*	*Two-level time slicing*
1.0	*	*
1.5	*	1.149
2.0	*	0.991
2.5	1.549	1.418
3.0	1.471	1.343
3.5	1.394	1.287
4.0	1.446	1.269
5.0	1.532	1.225

*Unstable queues

CPU dispatches in each run, as a measure of the relative overhead costs of the three strategies. The results are shown in Table 3. We see that the three strategies are close to each other in terms of total number of CPU dispatches. However, deadline scheduling results in markedly fewer page fetches than does two-level time slicing, which in turn has fewer than one-level time slicing. The low level of page fetching produced by deadline scheduling is probably due to the fact that the deadline scheduler revokes a page allocation only in the relatively rare event of some transaction reaching the urgency threshold, whereas the time slicing strategies revoke page allocations on every time slice end.

Conclusions

Given the particular environment on which our experiments were based, i.e., a mixed interactive and background load as characterized earlier, and neglecting contentions for resources other than CPU's and pages, the time slicing and deadline approaches to scheduling compared as follows:

1. Deadline scheduling can support more interactive users than time slicing.
2. Deadline scheduling results in faster average response than time slicing.
3. The three strategies studied are roughly equivalent as regards CPU dispatching overhead. However, deadline scheduling is markedly superior as far as paging overhead is concerned.

These results appear to corroborate the soundness of the underlying philosophy of deadline scheduling, which strives to make intelligent guesses at the true priorities of the various transactions in the system. The favorable response ratio distribution of deadline scheduling is evidence that the strategy successfully avoids delaying critical transactions in favor of less critical ones. The improved load-carrying capacity of deadline scheduling is explained by the reduced level of page fetching overhead of this strategy as compared with time slicing. Since the

Table 3 Overhead costs in 30-minute run (load = 48 interactive trans./min plus background).

Strategy	*No. of page fetches*	*No. of CPU dispatches*
Deadline scheduling	46104	5009
One-level time slicing (slice = 3.5 s)	58248	5084
Two-level time slicing (slice = 2.0 s)	51614	4994

"cost" of a page fetch is represented in the model by a 20-ms delay, the repeated page fetches induced by time slicing cause a reduction in system throughput.

These results must be considered preliminary, because the strategies and the modeling environment were considerably abstracted from their real counterparts. A number of avenues for further research suggest themselves, including an investigation of various ways of assigning deadlines to transactions, and the sensitivity of the deadline scheduler to the deadlines chosen. Another interesting extension to our experiments would be a study of the relationship between response ratio and CPU requirements for transactions in a heterogeneous transaction load, under the various scheduling strategies.

Appendix. User expectations of interactive systems

Pioneering work in measuring the behavior of interactive users was done by Scherr [10], whose analysis of the CTSS system at MIT revealed that on the average, a terminal user completes a transaction approximately every 36 seconds (35.2 s for I/O and user think time, plus 0.9 s of computation), and by Bryan [11], who showed that the "average" user of the JOSS system at RAND Corp. generated a transaction every 34 s (24 s thinking and typing request, 10 s for system to reply). Later studies were made by Fuchs, Jackson, and Stubbs [12,13], who consider that both user requests and system replies are composed of a "thinking" period followed by several "bursts" of characters. Their measurements on three time-sharing systems showed that an average user request was completed every 72.8 s (21.4 s to enter request, 51.4 s for system reply) [13] (amended in [12] to 57.3 s, 21.1 by user and 36.2 by system). Any attempt to interpret transaction rate in terms of number of users is complicated by the known fact that the transaction submittal rate of a user drops as the system response time increases. Since the results of our experiment are expressed as transaction rates, they are independent of this effect.

References and footnote

1. M. T. Alexander, *Time Sharing Supervisor Programs*, University of Michigan Computing Center, 1970.
2. A. J. Bernstein and J. C. Sharp, "A Policy-Driven Scheduler for a Time-Sharing System," *Communications of the ACM* **14**, 2, 74–78 (February 1971).
3. E. G. Coffman, Jr. and L. Kleinrock, "Computer Scheduling Methods and their Countermeasures," *Proc. 1968 SJCC*, 11–21.
4. W. J. Doherty, "The Effects of Adaptive Reflective Scheduling in TSS/360," *Proc. 1970 FJCC*, 97–111.
5. H. Hellerman, "Some Principles of Time-Sharing Scheduler Strategies," *IBM Systems Journal* **8**, No. 2 (1969).
6. B. W. Lampson, "A Scheduling Philosophy for Multiprocessing Systems," *Communications of the ACM* **11**, 5, 347–360 (May 1968).
7. R. Mahl, "An Analytical Approach to Computer Systems Scheduling," Ph.D. Thesis, Univ. of Utah, Salt Lake City, Utah (June 1970).
8. R. R. Muntz, and E. G. Coffman, "Pre-emptive Scheduling of Real-Time Tasks on Multiprocessor Systems," *Journal of the ACM* **17**, 2, 324–338 (April 1970).
9. The concept of deadline scheduling was first described in internal memoranda by Hans P. Schlaeppi.
10. A. L. Scherr, *An Analysis of Time-Shared Computer Systems*, The MIT Press, Cambridge, Mass., 1967.
11. G. E. Bryan, "JOSS: 20,000 Hours at a Console—A Statistical Summary," *Proc. 1967 FJCC*, 769–777.
12. E. Fuchs and P. E. Jackson, "Estimates of Distributions of Random Variables for Certain Computer Communication Traffic Models," *Communications of the ACM* **13**, 12, 752–757 (December 1970).
13. P. E. Jackson and C. D. Stubbs, "A Study of Multiaccess Computer Communications," *Proc. 1969 SJCC*, 491–504.

Received December 27, 1972; revised February 15, 1973

The authors are located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.