Fundamental considerations in time and space scheduling for time-sharing systems are reviewed. Workload components are classified as trivial and nontrivial foreground, and background. Each has certain resource-use and required response properties. A central issue in scheduling is the degree of advance knowledge available to the scheduler about calls on system resources. This provides a theme for classifying several algorithms.

A response figure of merit believed to be helpful in understanding time-sharing schedulers is defined. Simulation results using a very simple workload and system model are included in the discussion. A summary is given of some major issues in scheduling for time-sharing and virtual systems.

Some principles of time-sharing scheduler strategies

by H. Hellerman

A complete analysis of any data-processing system, including a time-sharing system, must consider two fundamental questions: (1) What functions are given the users of the system? (2) How are the resources that are used in implementing these functions controlled, allocated, and assigned? The word "functions" includes the number, convenience, and logical flexibility of the programming and control languages, the amounts and types of storage, and the provisions by which the user can modify and add to these facilities. In the second question, the set of considerations, called *scheduling*, is of primary interest in this paper, but it is not completely independent of the supplied functions. This is so because, as we shall see, a central scheduling issue is the degree of advance knowledge available on calls for resources. The degree is usually smaller, the greater the generality of functions provided.

scheduling

Scheduling is the process of assigning resources to a workload so as to satisfy some objective of "good" service. In computer system design, the objective is not usually stated initially as a precise mathematical function but eventually appears quite explicitly as part of the supervisor program. It is nevertheless conceptually useful to think of scheduling as a process of optimizing some objective function that is derived from the intended use of the system. In this paper, interest centers on time-sharing systems whose principal purpose is to give their users fast man-machine interaction. However, this objective should not be met

by excessive sacrifice of other desirable properties of data processing systems such as good problem-execution characteristics.

This paper examines fundamental considerations in the design of time-sharing schedulers. First, some basic considerations are outlined. Following that, certain scheduling variables and algorithms are classified, and a conceptual description of a rational design procedure is suggested. A simple, idealized model of a workload is then described, and some performance parameters, including an objective function, are defined. Simulation results are given for eight scheduler algorithms for a simple, single-server model. Finally, some important issues, results, and conclusions not included in the model are discussed.

Some basic considerations

The economic feasibility of a time-sharing system depends on its ability to service multiple users "concurrently," at least on the scale of human reaction times. Users are served by executing programs; some are written by the user, others are invoked by him but are supplied by the system. Although storage space can be shared by several programs, usually the central processing unit (CPU) cannot be shared, and it can service only one program at any instant. Thus all programs must share this resource in time. The desired concurrency cannot be ensured by sequential running of each program to completion before starting a new one because a long-running program could then unacceptably delay all those that follow. Most time-sharing systems therefore divide time into slices (quanta) and rapidly switch the CPU among the pending programs, giving each a time slice in some cyclic pattern, with due attention to new requests. For this process to succeed, programs must be interruptible.

Interruptibility is possible because the essential past history of any program is characterized by a *state*, which may be thought of as a string of bits that, if known at any time, completely determines the future logical properties of the program (together with future inputs). In a single processor system (assumed from now on), the state of at most one program, the active one, resides partly in processor registers and partly in main storage at any one time; the states of all other programs are, at that time, resident entirely in storage. Switching consists of "anesthetizing" the active program by storing the processor part of its state (processor registers) in main storage and then resetting the processor from the previously stored state of another program.

sharing and swapping

Of the potentially large number of connected users with pending requests for service, only a few (sometimes only one) can be held in the relatively small, expensive main storage at one time. The rest are kept on a slower, cheaper auxiliary storage drum or disk. Program execution can only be done directly from main storage, and as each program nears its next time slice, if it is not main-storage resident, it is exchanged (swapped) from auxiliary to main

storage with some program currently resident. Only the area of the replaced program modified since swap-in need be transmitted out since a copy of the nonmodified space already exists on auxiliary storage. Recognition of this can reduce the number of bytes to be transmitted, thus shortening swap time. Modern computers can switch and swap programs fast enough to satisfy the pseudo concurrency requirement.

user-system communication

Users communicate with the system through devices called *terminals*. Although there are many types of such devices, for our purposes a terminal may be envisioned as a typewriter-like device with several keys, one of which (e.g., the carrier-return) signals the system that the user has completed sending a message to the system. Striking this key is an example of an *interaction event*; another is when the system responds, e.g., by typing a message or unlocking the keyboard. Frequently, interaction events alternate between a user and the system in a "conversation." Such a pair will be called simply an *interaction*. A *tract* (abbreviation for *tra*nsac*t*ion) is defined as the work done for a single user during an interaction. The elapsed time to service a tract is called the *response time*. It depends on the nature of the tract, the system, and the activities of other users on the system. For scheduling purposes, all tracts are assumed to be independent.

Certain properties of tracts and the state of system resources are called *scheduling variables*. In the design of schedulers, these variables must be chosen and means specified for obtaining their values and operating on them to make decisions on resource assignments. Examples of scheduling variables for the workload include arrival time, explicit priority, time already expended, and expected-completion time for each tract. Scheduling variables for resources include busy, idle, or ready status and storage occupancy.

In addition to economical and effective man-machine interaction, conventional facilities such as language processing and problem program execution are also important. A basic design task is to rank the required services with respect to resources needed and with respect to sensitivity of performance to user satisfaction.

trivial response

User satisfaction depends critically on fast response to those tracts that arise from the most common human requests. Although there is no universal agreement as to what constitutes a complete set of such operations, certainly they must include entry, display, or modification of programs and data. It is fundamental and fortunate for the feasibility of a time-sharing system that each such operation usually requires only slight use of system resources so that many such concurrent requests can be serviced fast enough on the human time scale, even though they are processed one at a time on the same equipment. Because of their small use of resources, we shall call such requests *trivial* and the system response to them *trivial response*. A most important objective of a time-sharing scheduler is satisfactory trivial response not only to several concurrent trivial requests but also in the presence of heavy

nontrivial loads. The resource most critical to trivial response tends to be the speed of auxiliary storage holding information necessary to service imminent pending requests.

nontrivial response

Although ensuring fast trivial response is the first order of business of a time-sharing scheduler, the nontrivial tract initiated by the on-line user must also be processed with a responsiveness consistent with the interactive environment. Nontrivial tracts include program translation by the language processor(s) and the running of programs. The response requirement here is not as sharp as for trivial responses. However, a system directly servicing people will achieve success only by adequately meeting the users' expectations for good service—in this case, good response time. Human expectations are complex functions of many types of influences and change with time and experience. One principle which appears generally applicable is that human tolerance of response delay is (or can be made to be) roughly proportional to the complexity of the request, say, as measured by the amount of processing required to satisfy it. This suggests making priority-of-service correspond inversely to declared, expected, or estimated length-of-processing. This principle, applied to scheduling nontrivial work initiated from the system terminals, is also consistent with achieving fast trivial response.

Nontrivial tracts can call on all major resources provided to the user. The nature of these in turn define the generality of the system. All include arithmetic and program control functions usually implemented by a CPU. For a "dedicated" system, by which we mean here one requiring the same language of all users and typically restricting each to a fixed amount of main storage, the CPU is the critical resource. In more general systems where the user has the option of several language processors and programmed access to auxiliary storages and other devices, one of these devices or the channel controlling it often becomes the critical resource.

background

Many systems are imbedded in an environment where, in addition to the terminal users' trivial and nontrivial tracts, there is another component of workload with far less demanding response time requirements. It is natural and feasible to also process this workload on the same equipment used for time-sharing. *Background* is a type of workload that has nonstringent response characteristics. It is characterized by continual availability, say by batching. In fact, a principal source of background is the usual batched work of conventional systems. A prime requirement in time-sharing, not too difficult to satisfy, is that background service must not impair the foreground trivial response. Such impairment is avoidable because the weak response requirement permits deferment of background processing in favor of any new trivial request. The insensitivity to response time also permits scheduling of background during the frequent, but somewhat unpredictable, intervals when the CPU is not servicing foreground tracts. Note that the distinguishing feature of background work

is its low response-time sensitivity. It is feasible for background work (designated as such) to be initiated from a user terminal.

Another system sometimes linked with time-sharing is a virtual storage system. It supplies each user with a logical storage whose properties are independent of its implementation. The storage is an automatically managed hierarachy of (at least two) device types. Although virtual storage and time-sharing are independent ideas with no essential connection, they have been combined in a few announced systems (e.g., IBM TSS/360, GE-MULTICS, SDS Sigma 7). Such systems are now at the frontier of generality and present the most difficult scheduling problems.

A classification of the workload

The workload presented to a time-sharing system is now classified in a way that is pertinent to scheduling considerations. Each class of tract is summarized by four descriptors applicable to each tract of the class:

- Arrival characteristics
- CPU time requirements
- Auxiliary storage transmission
- Required response

Comparative terms such as "fast, few," are intended for rough comparison among the categories. Table 1 shows the classification.

The properties of trivial tracts and their critical importance to user satisfaction were discussed earlier and will not be treated further here.

Note the differences between background work and foreground work (terminal-initiated program execution). Background work is almost always "on-hand" whereas terminal-initiated requests arrive in an unpredictable pattern. Also, response times for

Table 1 Classification of components of system workload

Category of tract	*Arrivals*	*Required processor time*	*Auxiliary storage transmission**	*Required response*
Trivial	Unpredictable	Small-fixed Predictable	Small Predictable	Fast**
Terminal-initiated Nontrivial	Unpredictable	Variable Unpredictable	Variable Unpredictable	Variable
Background	Predictable (assuming some batching)	Variable Unpredictable Likely long	Variable Unpredictable	Slow

*Not including swapping.
**Fast and slow response are relative to human response times.

terminal-initiated programs, although necessarily variable over a wide range (depending on the nature of the tract and its demands for resources), should be better than the same tracts appearing in the background mode. The user submitting background work might well be given a billing advantage for his willingness to wait longer for results and for helping to keep a constant workload available to the system when time-sharing demand slackens. This policy follows analogous billing practices in the telephone and electric power industries, where rates are adjusted favorably for service during low-activity periods.

A classification of scheduling variables and algorithms

Some variables for each tract that may be used to "drive" a scheduler for a single-server model can be classified as follows: (1) explicit priority or deadline time, (2) arrival time (A), (3) execution time completed (P), and (4) total execution time (X).

Simple functions of these may also be used to obtain other scheduling variables.[1] With C used to denote "current-time," some examples are: (1) residence time ($C - A$), (2) time-in-queue ($(C - A) - P$), (3) remaining execution time ($X - P$), and (4) optimistic predicted deadline time ($A + X$). Those cases that include the variable X require advance knowledge of execution time.

Some scheduler algorithms using these variables are described in Table 2. This listing is similar to a recent one by Coffman.[2] One distinction introduced in the classification is a (Q) or (I) modifier. The former means that queue exploitation (switching to the next task) is done only after the tract currently in process is completed, i.e., the scheduler belongs to the *sequential* category. The (I) designation means that the scheduler can interrupt the current tract in process for queue exploitation—it can be done at either standard (time-slice) intervals and/or at each new arrival. Most time-sharing schedulers are type (I) algorithms (to be described later).

tie-breaking

The algorithms of Table 2 may be ambiguous in the sense that the selection rule may result in "ties," i.e., more than one tract with the same optimum value of the scheduling variable. Tie-breaking requires use of another rule among the ties; earliest arrival time is a common tie-breaking stratagem. However, this may also result in ties. In some systems, ties cannot occur due to physical restrictions on the arrival pattern. Where this is not so, an arbitrary method of assigning a unique tag to each tract can serve for ultimate tie-breaking.

In Table 2 the schedulers labeled 3a, 3b and 4a, 4b comprise pairs where the first member uses advance knowledge of execution time (X) and the second uses the simplest observed approximations of these, i.e., P for X. Simulation results for most of these schedulers for different workloads is given in the Appendix.

Table 2 Classification of some queue disciplines

	Scheduler name	*Abbreviation*	*Basic equation*	*Remarks*
0	Explicit priority			
1a 1b	First arrival, first service Last arrival, first service	FAFS (or FIFO) LAFS (or LIFO) LAFS(Q); LAFS(I)	$\lfloor/A$ $\lceil/A$	Priority depends on arrival time only.
2	Round robin	RR	$\lfloor/CLK$	CLK [I] is the time tract I last received a time slice.
3a 3b 3c	Shortest execution, first service Least completed, first service Least remaining, first service	SXFS(Q)*; SXFS(I)* LCFS(Q); LCFS(I) LRFS(Q)*; LRFS(I)*	$\lfloor/X$ $\lfloor/P$ $\lfloor/(X-P)$	Priority depends on execution time only.
4a 4b	Earliest deadline, first service Earliest estimated deadline, first service	EDFS(Q)*; EDFS(I)* EEDFS(Q); EEDFS(I)	$\lfloor/(A+X)$ $\lfloor/(A+P)$	Priority depends on both arrival and execution time.
5	Estimated f optimizer	EFMO	$\lfloor/P \div (C-A)$	

A Arrival-time vector
X Execution-time vector
P Execution-time-completed vector
(Q) Queue is exploited only after current-in-process task is completed (nontime-sliced)
(I) Queue is exploited shortly after arrival of new task (time-sliced)
* Execution (future) information is required
$\lfloor/$ Represents "minimum-of"
$\lceil/$ Represents "maximum-of"

It is not necessary for a scheduler to strictly use only one of these algorithms; a combination is readily possible. For example, the system may well have a good estimate for the execution time (X) of certain tracts, especially those in the "trivial" category where fast service is most critical.

design constraints

Advance knowledge of demands on resources is believed to be at the heart of scheduler design. This is also clearly indicated in the simulation results. Such knowledge can be made available at different times from different sources. For example, a good deal of information can be obtained once and for all at system-design time by limiting the possible resources available to the user. A "dedicated" system that limits all users to one language and language translator can have a simpler scheduler than one that must permit several languages and their translators. The simpler scheduler is possible partly because the single translator copy can be kept permanently resident in main storage and shared. This can appreciably reduce swapping time since program translators (compilers, interpreters) often account for much of the space required to service a user. In more general systems, sharing is still possible among those users requiring the same translator concurrently. An

added constraint, however, is placed on the scheduler. Specifying and enforcing conventions on subsystems to pass scheduling variables to the scheduler are difficult architectural and implementation problems for general-purpose systems.

Another illustration is the nonvirtual storage system that restricts each user to no more than some fixed fraction of real main storage. This constraint makes scheduling much simpler than the case where maximum main storage demand is unknown in advance as in virtual memory systems.

scheduler design procedure

If we are given a system with a set of constraints and are required to design a scheduler to meet some stated objective function, how can we proceed? It would be most helpful if we could first set down a scheduler, called a BEST scheduler, even though it requires advance knowledge of resource demands. This would give us a base for comparison of the performance of any practical scheduler and also be suggestive of a good scheduler. An orderly design procedure could then be envisioned as follows:

1. Define the objective function assuming advance knowledge of calls on resources (e.g., X) is available.
2. Devise a BEST scheduler algorithm to optimize the objective function.
3. Devise an algorithm for estimating the ranking of advance-knowledge variables (X) from observed variables (e.g., P). The estimated values are to be updated as the system runs.
4. Devise a scheduling algorithm that uses the estimates from item 3 in the scheduler derived in item 2.

The main requirement on the estimation process is that it yield a good approximation to the *ranking*, i.e., relative magnitudes of the scheduling variables, not the actual values of these variables. Although the process of scheduler design just described appears to be a rational one, it must be considered speculative. Most present practical schedulers are designed much less formally. A basic characteristic of an *adaptive* scheduler is its automatic monitoring and use of information on problem and resource states. The adaptive property is not an absolute, there being many degrees of it. Although there are no unique sufficient conditions for a scheduler to be called completely adaptive, a necessary property is that no operator intervention is required for scheduling.

An interesting adaptive scheduler is the one described for the MIT-7094 CTSS system.[3] It used an unoverlapped swapping strategy with a single user in main storage at a time. Scheduling is thereby simplified because there is no main-storage space allocation, only a CPU-time allocation. The basic idea of this scheduler is to give short-run-time tracts high priority for short CPU time slices. Run time was estimated for a starting tract from its main-storage size, but as a tract received one or more slices, its estimated relative length-of-run was in effect revised to correspond to the length-of-run already observed for that tract. Thus, as a tract proved itself longer and longer, the system automatically reduced its priority.

However, to reduce swapping overhead, the lower priority (longer) tracts, once selected, were given more time slices for their "shot." All runs were interruptible for newly arrived tracts which were entered into their proper position in the queue. For practical purposes, this adaptive scheduler thus treated all tracts in an automatically managed priority continuum.

Measures of scheduler performance

A simple model of a workload and a few performance measures are now defined largely independent of any particular scheduler. The values for the parameters are dependent upon both the workload and the scheduler.

The system workload model consists of tracts, each characterized by two numbers:

a_i = arrival time for tract i, (i.e., the time it is first considered for running by the system). In a system with typewriter terminals, a_i occurs upon the striking of the carrier-return key.

x_i = execution time for tract i

If the scheduler always allocates all available resources to the workload, it may be possible to measure x_i as the time to run the tract alone on the system. This is valid on most systems measured to date. However, there is at least one exception. The QUIKTRAN-style scheduler is not designed to optimize response to short (nontrivial) tracts, instead it is designed to approximately equal the response time on a given tract independent of the number of other tracts being processed concurrently. In this case, measured execution time on a tract running alone is much larger than x_i. Another case where run-alone time may be different from x_i is if the selected tract has several output phases, since then various types of output/compute overlap are possible, thus obscuring x_i.

With the above precautions in mind, we may define the following observable times:

q_i = time that tract i completes execution

$e_i = q_i - a_i$ = elapsed time for tract i

$w_i = e_i - x_i = q_i - (a_i + x_i)$ = wait or in-queue time for tract i

The term q_i is a "time-stamp" quantity measured from some common time origin for all i.

Some performance measures are:

Average elapsed time

$$\bar{e} = \frac{1}{n} \sum_{i=1}^{n} e_i \tag{1}$$

Average wait time

$$\bar{w} = \frac{1}{n} \sum_{i=1}^{n} (e_i - x_i) = \bar{e} - \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2}$$

We also seek to define a "figure-of-merit" which is to have the following properties:

figure-of-merit

- It should be dimensionless.
- It should have a maximum value of one for some ideal scheduler and system for *all* workloads.
- It should increase for a "better" system (unlike the average times above).
- It should be larger the more successful the system is in giving better service to short tracts relative to long ones in a stream containing both long and short tracts.

The last condition is only one of several possible, but is the one of interest in the remainder of this paper.

One parameter that appears to satisfy these conditions is

$$f = \frac{n}{\sum_{i=1}^{n} (e_i/x_i)} \tag{3}$$

In the language of statistics, f is the "harmonic mean" of x/e.

The case of $f = 1$ is found where each tract runs on a "private" system so that $e_i = x_i$. In other systems, a given elapsed-time value e_i will be weighted more *detrimentally* if it corresponds to a short run (small x_i) rather than a long one. The f values depend upon the workload (x_i, n), the computer and its programming support, and the scheduler.

Normalized reciprocal average wait time would also appear to satisfy most of these conditions, e.g., using Equation 2:

$$g = \frac{n}{\sum_{i=1}^{n} [(e_i - x_i)/x_i]} \tag{4}$$

By simple algebra, it is readily shown that g is trivially related to f as:

$$g = \frac{f}{1 - f} \tag{4a}$$

The term "response-figure-of-merit" hereafter refers to f (Equation 3). It is intended to be a sensitive measure of a system's ability to give higher priority service to short versus long tracts. It is therefore concerned with the relation of run times within a stream. The f function can be used in two ways: (1) as a way of "rating" a system and tract stream and (2) as an objective function for a scheduler algorithm.

Consider now the problem of comparing two different systems on the same tract stream. Measurements could be made of all x_i and all e_i. The f values could be computed for each system and then compared. It is easy to see that the f values cannot be the sole measures of system performance. As a simple example, suppose system A is 10 times faster than system B so that x_i and e_i for A will be 10 times smaller than corresponding values for B.

Equation 3 shows that f is identical for both systems because only ratios appear in the expression for f. System A, however, is capable of roughly 10 times the work as B is in the same time. We must look to a statistic of performance other than f to find a measure of this advantage.[4]

throughput

Throughput is defined for a given tract stream as the reciprocal average time to complete all tracts in the stream, i.e., if q_n is the time the last tract is completed, measured from the arrival time of the first tract, throughput is

$$t = \frac{n}{q_n} \tag{5}$$

Throughput, unlike f, is insensitive to the order in which tracts are executed, i.e., a system that processes long tracts before short ones can have the same t value as one that schedules short ones early. However, in the former case, the response "felt" by the user (and f) would be much poorer.

Complete comparisons of the performance of two systems require both f and t for the same set of tract streams on both systems. A scheduler designed to directly optimize f of Equation 3 would select for next-time-slice, that tract with the largest e_i/x_i value, i.e., the smallest $x_i/(q_i - a_i)$. This strategy is, however, not practical since both x_i and q_i are not known in advance. Simple estimates of these, obtainable from observations during system operation are: p_i for x_i and current time (c) for q_i. Such a scheduler will be called an "estimated figure-of-merit optimizer" and is designated EFMO in Table 2.

Sequential and round-robin schedulers

sequential scheduler

Consider first the very simple situation where all tracts to be serviced are available at the same time; service of tract i requires a total of x_i time units from a single server. If the scheduler is *sequential*, i.e., it runs each tract to completion before considering another, the elapsed time for job i is

$$e_i = \sum_{j=1}^{i} x_j \tag{6}$$

From Equations 1 and 2, average elapsed time ($\bar{e}$) and average wait time ($\bar{w}$) are

$$\bar{e} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{i} x_j \tag{7}$$

$$\bar{w} = \bar{e} - \frac{1}{n} \sum_{i=1}^{n} x_i \tag{8}$$

From Equations 3 and 6, the response figure-of-merit is

$$f = \frac{n}{\sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \Big/ x_i \right)} \tag{9}$$

The above equations may be applied to the case of nonidentical

arrival times during all intervals when at least one tract requires service. One common sequential scheduling rule is earliest-arrival or first in first out (FIFO) which schedules that tract for next service whose arrival time to the system is the smallest. The ordering of the indices i and j in the above equations then corresponds to the ranking of arrival times with some tie-breaking rule whenever two or more tracts arrive at the same time.

BEST scheduler

Another sequential scheduler of great theoretical interest assigns next service to that tract whose execution time (x_i) is the smallest. The scheduler then must be capable of ranking x_i and ordering the tracts so that their execution times, now called x', is a permutation of the original x with the rule

$$x'_i \leq x'_{i+1} \quad \text{for all } i \tag{10}$$

For the case of identical arrival times, the algorithm may be shown to maximize the response figure-of-merit f and is therefore called BEST. It could be modified to accommodate staggered arrivals by reranking the x'_i whenever new arrivals occur, and if necessary preempting the tract that is in service with the one holding the best rank. A fundamental problem remains however; the BEST algorithm requires advance knowledge of execution times that are often not available. Despite this difficulty, BEST is suggestive of practical schedulers and can also serve as a base for comparison of all schedulers.

round-robin scheduler

Sequential schedulers suffer from the defect that since execution is not interruptible, and execution times are not in general known in advance, long-run tracts may delay service to short ones resulting in a poor f and, hence, by our criterion, poor service. The *round-robin* scheduler prevents this by allocating one time-slice successively to each pending tract, cycling back to the first after the last has received its slice. This process may be described in another way: record the time each slice is given to each tract; select for next service that tract with lowest such recorded value. A recording of zero for new arrivals gives them high priority for their first slice. This policy is consistent with good service to trivial requests since they are frequently serviced in one time slice (or less).

Neglecting new arrivals, we may set down an approximate analytic relation for the elapsed time of tract i for a round-robin scheduler if we assume that the round-robin cycle is in the order of shortest tract first (the best case). Although this appears to be a drastic assumption, round-robin schedulers are relatively insensitive to the order of the next slice. With this assumption, elapsed time is

$$e'_i = (n - i)(x'_i - 1) + \sum_{k=1}^{i} x'_k \tag{11}$$

Substituting Equation 11 into Equation 1 gives the average elapsed time as

$$\bar{e}' = \left(\sum_{i=1}^{n} \left(1 - \frac{i}{n}\right) x'_i\right) - \left(\frac{n-1}{2}\right) + \frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{i} x'_k \tag{12}$$

The right-most term can be identified with Equation 7 and, hence, gives the best average elapsed time for a sequential scheduler. The remaining terms on the left result in a nonnegative value. Equation 12 thus shows mathematically that the round-robin scheduler gives an average elapsed time that is never better than that of the BEST sequential scheduler (and is in most cases worse). An intuitive reason for this result is as follows: to complete i tracts, the BEST sequential scheduler worked only on the i tracts, but the round robin did the same and also gave some time to all others. Note that these properties follow only from the sequencing properties of the schedulers and are not due to "overhead," i.e., use of resources for system control. Overhead, however, tends to reinforce the above characteristics.

To illustrate the comparison of sequential and round-robin scheduler strategies, a simple set of examples was constructed, and the performance parameters computed as shown in Table 3. Each example consists of a specified stream, all tracts available at the same time (all $a_i = 0$), and each is characterized by a run time when run alone (x_i). The time to run each stream is about equal to that to run any other. Average user wait time and the figure-of-merit f is given for each stream for the FIFO sequential, round-robin, and BEST sequential algorithms.

For more on simulation results, see the Appendix.

Resource states as scheduling variables

Emphasis thus far has been on those scheduling variables that characterize the tracts. We turn now to the influence of resource states on scheduling. The principal factors of this type are: CPU utilization, main storage occupancy, and auxiliary storage access state.

output terminal buffering

As a first simple example, consider a user's program request for data typeout. In many systems, CPU scheduling for this user is suspended until the output is completed although allocation of time slices continues for other users. In other words, there is overlap of output typing and computation between users but not for the same user. However, some systems (e.g., APL[5]) also overlap a user's output with *his* computation. Nontrivial response can thereby be improved appreciably for applications that alternate computation and typeout. These include many formula evaluation and simulation applications as well as program traces valuable in on-line debugging. Such programs can often appear to the user to run at nearly output typing speed.

background processing

A time-sharing system serving only terminal (foreground) users and near the limit of acceptable performance may typically show an appreciable fraction of CPU idle time. However, this inactivity is distributed over short unpredictable intervals. If an attempt is made to fill this time by admitting more foreground users, there is little probability that their unpredictable demands

Table 3 Illustrative tract streams for sequential and round-robin schedulers

		Algorithms									
		Sequential (FIFO)			*Round robin*			*BEST*			*Throughput***
Code	*Tract* streams execution times (X)*	*Average elapsed time*	*Average wait time*	*f*	*Average elapsed time*	*Average wait time*	*f*	*Average elapsed time*	*Average wait time*	*f*	
1	5, 5, 5, 5, 5, 5, 5, 5, 5	25.0	20.0	0.200	41.0	36.0	0.122	25.0	20.0	0.200	0.200
2a	9, 9, 9, 9, 2, 2, 2, 2, 2	33.3	28.2	0.078	28.7	23.6	0.151	17.8	12.7	0.306	0.196
2b	2, 2, 2, 2, 2, 9, 9, 9, 9	17.8	12.7	0.306	26.4	21.3	0.181	17.8	12.7	0.306	0.196
3a	9, 8, 7, 6, 5, 4, 3, 2, 1	31.7	26.7	0.082	31.7	26.7	0.143	18.3	13.3	0.333	0.200
3b	1, 2, 3, 4, 5, 6, 7, 8, 9	18.3	13.3	0.333	27.7	22.7	0.193	18.3	13.3	0.333	0.200
4a	9, 8, 7, 6, 3, 3, 3, 3, 3	30.6	25.6	0.118	32.8	27.8	0.139	19.4	14.4	0.281	0.200
4b	6, 7, 8, 9, 3, 3, 3, 3, 3	29.4	24.4	0.122	32.1	27.1	0.141	19.4	14.4	0.281	0.200
4c	9, 3, 8, 3, 7, 3, 6, 3, 3	28.1	23.1	0.136	32.1	27.1	0.144	19.4	14.4	0.281	0.200
4d	6, 3, 7, 3, 8, 3, 9, 3, 3	25.9	20.9	0.152	31.4	26.4	0.146	19.4	14.4	0.281	0.200
4e	3, 3, 3, 3, 3, 9, 8, 7, 6	20.6	15.6	0.259	30.6	25.6	0.155	19.4	14.4	0.281	0.200
4f	3, 3, 3, 3, 3, 6, 7, 8, 9	19.4	14.4	0.281	29.9	24.9	0.158	19.4	14.4	0.281	0.200

* All tracts arrive at same time
** Throughputs nearly identical for all streams

will match the periods of idleness, and there is considerable danger that critical response times will become unacceptable. The idle capacity can, however, be used without this hazard if it is applied to the background class of workload.

The scheduling of background involves several, often conflicting, factors. As was previously stated, work on background must not degrade trivial response, and this condition can be satisfied. If background is to be scheduled during short unpredictable CPU idle periods, it should be resident in main storage where it is always immediately accessible.[6] Such residence is essential if the CPU is always to be scheduled during swapping of foreground tracts. Another argument for permanent background residence is that such jobs are usually permitted access to input/output devices that can involve long noninterruptible transmissions requiring appreciable main storage space for their duration. Although the cycle-stealing organization of most modern systems[7] permits these to proceed concurrently with CPU activity, the main storage allocation must be maintained, and this area is not available for swapping during these times. Incidentally, this unavailability can also arise from foreground jobs if they are permitted unrestricted access to certain input/output devices. In either case, such activity need not inhibit swapping of foreground users provided that distinct space is used for both functions and that they do not conflict on some other facility such as a common channel or disk access mechanism. The design issue should now be clear: best concurrency, and, hence, time performance, requires a permanent area of main storage for background, but this denies both background and foreground users the maximum possible fraction of main storage space.

In contrast to the space allocation problem, assignment of CPU time to background work is a somewhat less complex problem. First call on CPU time slices should be to any pending trivial tracts and the first time slice of new foreground requests. Beyond this, in the event of a queue of nontrivial foreground tracts, the proportion of time slices given to these and to background is largely a matter of throughput and gross priority considerations. It is probably best determined by each installation according to its particular objectives.

The reservation of a fixed area of main storage for use by background is one important factor of main storage allocation. There are other space scheduling options open to designers that can also have profound effects on system function and performance. Several methods of main storage management are shown in Figure 1.

overlapped swapping

A system with only one foreground user's area in main storage can make all available main storage space accessible to each user. Such a system cannot overlap swap time of one foreground tract with the CPU time of another since overlap requires that an appreciable part of available main storage be allocated to receive or send the transmitted information. Thus the desirable time

efficiency of overlapped swapping is dichotomous with the benefits of largest possible user storage size.

complementary replacement

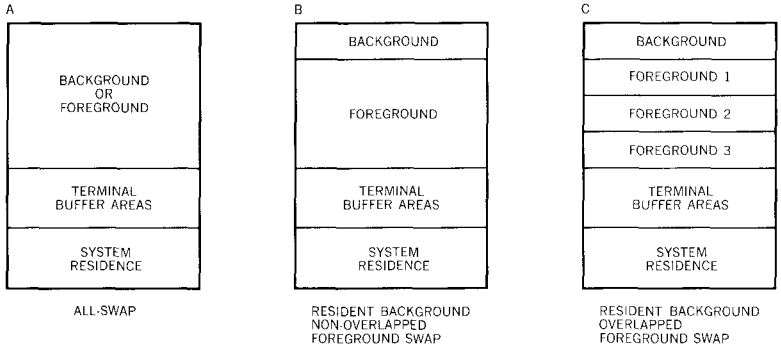
If it is decided to partition main storage into several user areas, a replacement problem may now arise: if the tract selected for next service is not main-storage resident, how should one of the tracts currently resident be selected for swap-out to make room for the incoming tract. One simple, intuitively appealing strategy will be called the Principle of Complementary Replacement. It is based on the idea that if a scheduling rule R is used to make the *best* choice of tract to receive the next time slice, this rule can also be applied to find the tract that is the *worst* choice. This "complementary" rule, called R^c, is applied to only those tracts currently resident in main storage, and the tract found by R^c is the one selected to be replaced. This principle appears to be applicable to several scheduling algorithms and makes replacement a simple variant of the service selection scheduling algorithm.

The choice of the tract to receive the next time slice can be done according to various criteria as described earlier in this paper. It can also include the effects of transmission (swapping) delays, e.g., by giving some priority weight to those tracts currently in main storage or closest to the current access position on the auxiliary storage device.

Paging systems

The critical role of main storage allocation has prompted fundamental studies of program needs for this resource. These studies have shown that the space viewed by a programmer in writing a program, called the *address space*, is often far larger than the space referenced in any one run of the program. In addition to overestimation due to oversight or desire to accommodate a range of data volumes, it is due to such unpredictable effects as large areas of program or data space that are not reached in a particular run because their reference is dependent on computed values that are not encountered. Even considering only space referenced at

Figure 1 Three methods of main storage management



least once, most programs show a high "locality of reference," i.e., the tendency to dwell for appreciable time periods in a few small areas of total address space. The areas are not generally contiguous with each other. The extent of these properties depends on the nature of the job and the style in which it is programmed. Yet they seem sufficiently common to encourage design of systems based on them. Such a system usually partitions both address and main storage space into fixed blocks called *pages*.

pages

A page is the unit of storage allocation used by the system but is hidden from the user. Since pages resident in main storage belonging to a given program at any instant may represent any part of address space, a hardware mapping device is provided to translate each program address (referencing address space) to a main storage address. This organization permits main storage to be treated as a pool of pages for allocation purposes, thus reducing the wasteful effects of contiguity constraints and, even more importantly, permitting the system to respond to actual program needs rather than worst-case estimates. In such a system there is little logical reason to restrict the user to an address space smaller than the main storage physical space; the address space is therefore often larger, hence the term "virtual storage" applied to the address space. The potential advantages of the paged allocation method include: better storage management based on actual demands, programmer convenience in having a large virtual store and, since address space and physical space are now logically separated, compatibility of programs across main storage sizes.

If we confine our attention for the moment to a single program whose address space is larger than main storage, it may well happen that after the main store is filled with pages, one of these executes a reference to a page not currently resident. The system must now use a replacement rule to decide which page to transmit to auxiliary storage (drum/disk) to make room for the new request. Although this process has some similarity to swapping, virtual storage page scheduling is a more complex problem since the system has little advance information of page requests, and the volume of page status information can be quite large. Excessive replacement means heavy paging and accompanying transmission delays. A number of replacement algorithms have been studied.[8] They seem to show a surprising lack of consistent favor to any one replacement rule assuming the same rule must be used over a set of programs.

threshold phenomenon

Experimental study of several programs, each running alone (nonmultiprogrammed) on a virtual storage system, reveals the following phenomenon:[9,10]

> Most programs may be characterized by a "threshold" size of physical storage, in general different for different programs which, if not available, tends to result in a paging "explosion," i.e., a very sharp increase in paging activity and resultant drop in performance compared to the case where above-threshold

size is available. Put another way, if the running of a program is attempted in less than its threshold space, the program generates very frequent page demands as it attempts to expand to threshold size.

For example, the following numbers are not atypical: a 20 percent decrease in main storage size below the threshold size resulted in a *factor* of 10 degradation in run time of a certain program compared to when threshold size was available. The extent of this phenomenon, of course, depends on the particular program being run. The program in turn depends in part on programming style. There is some evidence that if the programmer (including the systems programmer) observes a few simple guidelines, he can considerably soften the paging explosion problem.[9,10] As an example, he should organize his program for good "locality-of-reference" by keeping successively executed storage references in as few areas of address space as possible.

performance protection

The threshold phenomenon of the virtual storage type of system has a number of implications to scheduler design. To help understand these, it is well to state a general principle—classes of functions supplied to users should be carefully ranked with regard to required response sensitivity, and the scheduler should ensure that no matter what the load in a given category, it must have minimal effect on response times in all more sensitive categories. We shall call this principle, which has already appeared throughout this paper, the "performance-protection policy."

In the application of this principle to paged virtual systems in a time-sharing environment, a primary requirement is that no matter what set of thresholds may be present, trivial response times (the most sensitive category) must not be significantly impaired. A second application of the performance-protection policy may be applied to another serious problem: the program(s) with a sharp threshold whose threshold main storage size is larger than the machine's main storage. Such a program will inevitably run slowly, but the policy says that it must not be permitted to appreciably slow the running of smaller programs. More commonly, the system will have several requests pending, each with threshold size smaller than total main storage, but with their aggregate larger than main storage. The pool of physical pages must be allocated to the page demands of contending programs in such a way that minimum time is spent unproductively by programs attempting execution when less than their threshold main storage size is available to them. The major problem here is to prevent programs from crowding each other in main storage with resulting page-demand explosion.

R. W. O'Neil devised a rather simple but effective method called *load-leveling* to prevent page explosion due to competition between programs.[11] The idea is to observe the conjunction of two events easily monitored by the supervisor program: heavy paging and low CPU utilization. When these occur together, the supervisor

reduces the multiprogramming level by temporarily removing one of the programs from main storage contention, freeing its space for use by the other programs. This scheme gave very substantial improvement over a similar system on the same workload operating without the load leveler.

Concluding remarks

Scheduling depends on the demands for resources generated by the tracts and on the state of the resources, especially the space occupancy of the storages. Scheduling is simpler and can more nearly optimize response time and use of resources if a high degree of advance knowledge of calls on the resources is available. The amount of such advance knowledge tends to vary inversely with the generality of the functions provided. Limiting the user to a single language and fixed main storage size, while functionally restrictive, gives the scheduler much advance knowledge on calls on resources. It is relatively easy to achieve high scheduling efficiency in such a system.

The performance of a general system has often been poorer than a single-language ("dedicated") system, assuming the comparison is made on equivalent equipment with a workload both can execute. At this state of the art, it is not clear what part of this difference, especially on trivial and short nontrivial tracts, is *inherently* tied to system generality and what part is due to the fact that we presently know far more about the design of dedicated systems than about general time-sharing systems. Recognizing the reasons why it is difficult to achieve high performance in a very general system is not the same as believing that this performance penalty is inherent in the system's generality and that simple workloads must *necessarily* be treated poorly by such a system. In the opinion of this author, when design technology matures, a general-purpose system should *not* show significantly poorer performance on simple workloads than less general systems.

Appendix: Some simulation experiments

A simulator program has been written in the APL/360 language to simulate a simple workload model on a single-server system model for several schedulers. The workload model assumes each tract to be characterized by two numbers a_i and x_i (arrival and execution times) as described in the section on measures of scheduler performance. The system model permits a maximum of MXM tracts to be main-storage resident concurrently and has a one-way swap time of ST time units. The time unit throughout the simulator is the time slice. The simulator is capable of evaluating the effects on response times and figure-of-merit of: (1) various orderings of the tract stream, (2) arrival and execution-time differences, (3) eight scheduler algorithms, (4) swap time, (5) maximum number of users permitted in main storage concurrently, and (6) overlap or

nonoverlapped swap option. Each scheduling rule is used to determine which tract is to receive the next time slice. If this tract is in main storage, it is serviced. If not, and there is space in main storage, it is entered. In this case, if overlap is specified by the user, during the input operation a resident tract is serviced. If the tract selected is not in main storage, and there is no space in main storage, the complement replacement rule (see text) is used to decide which tract is to be swapped out. If overlap has been specified, during such a swap-out a resident tract will be serviced.

Although eight specific scheduler algorithms are supplied, others may easily be added.

In the simulations cited here, arrival and execution times for the tracts are specified explicitly, but random selections using a distribution function to generate the **A** and **X** vectors could be added to the simulator. All of the scheduler algorithms of Table 2 except LIFO are included in the program.

Table 4 shows the statistics obtained for one artificial tract stream with eight scheduler algorithms and systems that can accommodate 1 and 2 users in main storage concurrently. One set of statistics is for zero swap time, whereas the others are for a system with one-way swap time equal to the time slice.

Tables 5A and 5B show simulation results using execution-time data of a stream of 15 real FORTRAN execution jobs. In this experiment, the effect of different orderings of the tracts (even though they all arrive at the same time) was investigated. Although $15! = 1.31 \times 10^{12}$ orderings are possible, only three—the given one, the best, and the worst, were simulated.

Some conclusions from the simulation results are as follows:

1. Under zero swap-time conditions, the best f value of 0.508 was achieved by the advance-knowledge schedulers SXFS and LRFS. The best of the more practical class was LCFS followed closely by RR (0.298 and 0.290).
2. With nonzero swap time, and only a single tract permitted in main storage, the f values for LCFS and RR dropped to 0.067. This improved slightly (to 0.075) if two tracts were permitted in main storage concurrently. By also permitting overlapped swapping, LCFS and RR figure-of-merit rose to 0.137 and 0.109, respectively. In this experiment, overlapped swapping improved f by almost a factor of two, but this still was about three times poorer than a zero swap-time system.
3. Throughput was affected more by overlap than was the f measure of performance. The zero swap-time throughput was 0.214 for all schedulers. With swap time of one and no overlap, RR dropped to 0.08 while FIFO dropped only to 0.15. However, with overlapped swapping, the time-slice schedulers improved appreciably (e.g., RR throughput rose to 0.200).
4. Even with zero swap time, the ordering of tracts within the same stream can make a substantial difference in the f (and hence, response) as seen in Tables 3 and 5.

Table 4 Simulator results for eight schedulers on four configurations using one tract stream

Code	Scheduler		*Figure-of-merit f*				*Reciprocal average elapsed time*			
			MXM = 1		ST = 1; MXM = 2		MXM = 1		ST = 1; MXM = 2	
			ST = 0	ST = 1	OVLAP	NOVLAP	ST = 0	ST = 1	OVLAP	NOVLAP
1	Earliest arrival	(FIFO)	0.056	0.039	0.047	0.038	0.032	0.023	0.026	0.022
2	Round robin	(RR)	0.290	0.067	0.109	0.075	0.055	0.016	0.038	0.019
3	Least completed	(LCFS)	0.298	0.070	0.137	0.077	0.057	0.017	0.048	0.020
4	Earliest estimated deadline	(EEDFS)	0.118	0.038	0.086	0.053	0.041	0.014	0.035	0.019
5	Shortest execution	(SXFS)*	0.508	0.162	0.183	0.179	0.086	0.041	0.048	0.043
6	Least remaining	(LRFS)*	0.508	0.162	0.181	0.179	0.086	0.041	0.049	0.043
7	Earliest deadline	(EDFS)*	0.356	0.115	0.124	0.123	0.075	0.037	0.042	0.038
8	Estimated *f* optimizer	(EFMO)	0.254	0.067	0.139	0.075	0.054	0.016	0.049	0.019

Code	Scheduler		*Throughput*				*Number of half swaps*		
			MXM = 1		ST = 1; MXM = 2		MXM = 1	ST = 1; MXM = 2	
			ST = 0	ST = 1	OVLAP	NOVLAP	ST = 1	OVLAP	NOVLAP
1	Earliest arrival	(FIFO)	0.214	0.150	0.182	0.150	24	24	24
2	Round robin	(RR)	0.214	0.080	0.200	0.100	94	40	64
3	Least completed	(LCFS)	0.214	0.088	0.200	0.103	80	34	60
4	Earliest estimated deadline	(EEDFS)	0.214	0.087	0.200	0.111	82	38	52
5	Shortest execution	(SXFS)*	0.214	0.146	0.200	0.150	26	24	24
6	Least remaining	(LRFS)*	0.214	0.146	0.207	0.150	26	24	24
7	Earliest deadline	(EDFS)*	0.214	0.146	0.200	0.150	26	24	24
8	Estimated *f* optimizer	(EFMO)	0.214	0.080	0.200	0.100	94	32	64

* Denotes scheduler that uses advance knowledge of execution time
MXM Maximum number of tracts in main storage
ST Half (one-way) swap time in units of time-slice
OVLAP Overlapped swapping
NOVLAP No overlapped swapping

TRACT STREAM USED													
ARRIVAL TIMES:	A	0	1	1	1	5	5	5	10	11	11	11	11
EXECUTION TIMES:	X	20	1	5	1	1	10	1	5	1	5	5	1

Table 5A Gross simulation results using data of 15 FORTRAN tracts: Three orderings of the same tract stream

Scheduler	Code*	*Average elapsed time*			*Average wait time*			*Figure-of-merit f*		
		G	W	B	G	W	B	G	W	B
Sequential (FIFO)	1a	520	829	232	453	762	166	0.013	0.006	0.354
Round robin	2	395	398	391	328	332	325	0.112	0.103	0.130
Least completed	3b	395	398	391	328	332	325	0.111	0.103	0.130
Shortest execution	3a	232	232	232	166	166	166	0.354	0.354	0.354
Least remaining	3c	232	232	232	166	166	166	0.354	0.354	0.354

* Scheduler codes are as defined in Table 2

Table 5B Gross simulation results using data of 15 FORTRAN tracts: Three orderings of the same tract stream

GIVEN ORDERING (G)	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	X	240	106	39	19	50	3	3	3	10	4	1	137	138	42	200
WORST ORDERING (W)	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	X	240	200	138	137	106	50	42	39	19	10	4	3	3	3	1
BEST ORDERING (B)	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	X	1	3	3	3	4	10	19	39	42	50	106	137	138	200	240

Figure 2 Detailed simulator output for three schedulers

```
      SIMULATE
ENTER 1 TO PRINT GROSS STATISTICS ONLY, 0 TO PRINT FULL STATISTICS
□:
      0
ENTER SCHEDULER CODE LIST; 0 WILL DISPLAY CODES AND NAMES
□:
      1 2 5
ENTER 0 TO USE EXISTING  ARRIVAL, EXECUTE, AND SYSTEM PARAMETERS, ENTER 1 TO CHANGE PARAMETERS
□:
      1
ENTER THE PARAMETERS MXM ST AND V DEFINED AS FOLLOWS ON 1 LINE
 MXM= MAX. NO. TRACTS IN MAIN STORE; ST= ONE WAY SWAP TIME; V= 1 OR 0 (OVERLAP OR NO OVERLAP)
□:
      2 1 1
ENTER ARRIVAL AND EXECUTE TIMES FOR EACH TRACT ON 1 LINE; ENTER 0 0 TO TERMINATE
     ARRIVAL  EXECUTE
□:
      0 5
□:
      1 1
□:
      1 1
□:
      1 1
□:
      3 5
□:
      3 1
□:
      0 0
SCHEDULER CODE =1: FIRST-IN, FIRST OUT (FIFO)
ARRIVAL,EXECUTE(ALONE),COMPLETION,ELAPSED AND WAIT TIMES PER TRACT:

   0   1   1   1   3   3
   5   1   1   1   5   1
   7   8  11  12  19  20
   7   7  10  11  16  17
   2   6   9  10  11  16
SCHEDULER CODE =2: ROUND-ROBIN (RR)
ARRIVAL,EXECUTE(ALONE),COMPLETION,ELAPSED AND WAIT TIMES PER TRACT:

   0   1   1   1   3   3
   5   1   1   1   5   1
   9   4   7  10  18  14
   9   3   6   9  15  11
   4   2   5   8  10  10
SCHEDULER CODE =5: SHORTEST EXECUTION TIME (SXPS)
ARRIVAL,EXECUTE(ALONE),COMPLETION,ELAPSED AND WAIT TIMES PER TRACT:

   0   1   1   1   3   3
   5   1   1   1   5   1
   9   4   7  10  18  13
   9   3   6   9  15  10
   4   2   5   8  10   9
MAX. NO. TRACTS IN MAIN STORE=2  ONE HALF SWAP TIME(UNITS OF TIME SLICE) = 1 OVLAP=1
SCHED CODE       FIG. OF MERIT     1*AVE. ELAPSED   THRUPUT          NO. HALF SWAPS   AVE. WAIT

   1                0.121            0.0882          0.3              12               9
   2                0.1775           0.1132          0.3333           12               6.5
   5                0.1829           0.1154          0.3333           12               6.3333
CPU TIME(SEC.) FOR SIMULATION IS: = 38.93333333
```

The preceding conclusions are drawn from only a few of all possible workload cases. Other cases may be investigated with the simulator. Figure 2 shows the output from a typical simulator session. The simulator prompts the user to specify the needed parameters. He can also specify print options: gross statistics only or full statistics.

ACKNOWLEDGMENT

Many of the ideas in this paper were sharpened by discussions with H. J. Smith, Jr.

CITED REFERENCES AND FOOTNOTE

1. R. W. Conway, W. L. Maxwell, L. W. Miller, *Theory of Scheduling*, Addison-Wesley Publishing Company, Reading, Massachusetts (1967).
2. E. G. Coffman and L. Kleinrock, "Computer scheduling methods and their countermeasures," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **32**, 11–21 (1968).
3. P. A. Crisman (Editor), *The Compatible Time-Sharing System*, (2nd Edition) MIT Press, Cambridge, Massachusetts (1965).
4. Y. Ron has suggested that by defining x_i as the execution time alone of a common "base" system, f defined by Equation 3 can then include some throughput effects.
5. L. M. Breed and R. H. Lathwell, *APL/360*, IBM Contributed Program Library, 360D-03.3.007, International Business Machines Corporation, Program Information Department, Hawthorne, New York (1968).
6. M. Tsujigado, "Multiprogramming, swapping and program residence priority in the FACOM 230–60," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **32**, 223–228 (1968).
7. H. Hellerman, *Digital Computer System Principles*, McGraw-Hill Book Company, New York, 124–125 (1967).
8. L. A. Belady, "A study of replacement algorithms for a virtual storage computer," *IBM Systems Journal* **5**, No. 2, 78–101 (1966).
9. B. Brawn and F. Gustavson, "Program behavior in a paging environment," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **33**, 1019–1032 (1968).
10. E. G. Coffman and L. C. Varian, "Further experimental data on the behavior of programs in a paging environment," *Communications of the ACM* **11**, No. 7, 471–474 (July 1968).
11. R. W. O'Neil, "Experience using a time-shared multiprogramming system with dynamic address relocation hardware," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **30**, 611–621 (1967).

GENERAL REFERENCES

1. G. E. Bryan, "JOSS: 20,000 hours at a console, a statistical summary," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **31**, 769–777 (1967).
2. E. F. Codd, "Multiprogram scheduling," *Communications of the ACM* **3**, No. 6, 347–350 (June 1960) and **3**, No. 7, 413–418 (July 1960).
3. G. H. Fine, C. W. Jackson, and P. V. McIsaac, "Dynamic program behavior under paging," *Proceedings of the 21st National Conference of the ACM* **P-66**, 223–228 (1966).
4. D. N. Freeman and R. R. Pearson, "Efficiency vs responsiveness in a multiple services computer facility," *Proceedings of the 23rd National Conference of the ACM* **P-68**, 25–34B (1968).
5. D. H. Gibson, "Considerations in block-oriented systems design," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **30**, 75–80 (1967).
6. L. Kleinrock, "A conservation law for a wide class of queuing disciplines," *Naval Research Logistics Quarterly* **12**, No. 2, 181–192 (June 1965).
7. B. W. Lampson, "A scheduling philosophy for multiprocessing systems," *Communications of the ACM* **11**, No. 5, 347–360 (May 1968).

8. N. R. Nielsen, "An approach to the simulation of a time-sharing system," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **31**, 419–428 (1967).
9. A. L. Scherr, *An Analysis of Time-Shared Computer Systems,* MIT Press, Cambridge, Massachusetts (1967).
10. J. L. Smith, "An analysis of time-sharing computer systems using Markov models," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **28**, 87–95 (1966).
11. D. F. Stevens, "On overcoming high-priority paralysis in multiprogramming systems: a case history," *Communications of the ACM* **11**, No. 8, 539–541 (August 1968).