BELLCOMM, INC.
955 L'ENFANT PLAZA NORTH, S.W. WASHINGTON, D. C. 20024

B69 06024

SUBJECT: Accounting of Computer System Use in EXEC 8 - Case 803

DATE: June 5, 1969

FROM: A. L. Rothstein

ABSTRACT

Multiprogramming systems should be required to log in detail all user activity because each user is willing to pay only for what he uses. This capability is easily extended to logging system characteristics for purposes of analysis of system behavior.

The UNIVAC EXEC 8 system logs only CPU time for each run. This is largely irrelevant. It is impossible to use this time estimate as a basis for priority assignment (as is done in EXEC II) since the impact of a run on system throughput is dominated by its core requirements and I/O characteristics as well as CPU time.

We have modified EXEC 8 to log the number of I/O operations as well as the core-time product used by the run. In computing core-time a run is assessed 80 milliseconds of elapsed time for every I/O operation. Let C be the core-seconds (8192 words of core for one second), T the CPU time, and I the number of I/O operations. Charge $\simeq \frac{C}{60} + \frac{T}{30} + \frac{I}{1000}$. The time field on the run card is used to estimate the charge and is the basis of priority assignment. The charge is biased against core use tc induce greater use of the less critical facilities of CPU and I/O. Because of this bias, charge/hour processed by the system cannot be used as a measure of system throughput.

The current implementation charges the same for all devices and makes no allowance for concurrent operations. Thus, the user has no measure for the effect of changing file residence and multiprogramming within his own run and certainly no inducement to do so. We are suggesting a fairly simple modification to correct this deficiency.

The accounting system should be extended to include a charge for every service provided to the user.

BELLCOMM, INC.
955 L'ENFANT PLAZA NORTH, S.W. WASHINGTON, D. C. 20024

SUBJECT: Accounting of Computer System Use In EXEC 8 - Case 803

DATE: June 5, 1969

FROM: A. L. Rothstein

MEMORANDUM FOR FILE

I. Introduction

Accounting for computer use in batch systems is a straightforward process. Only one user program can execute at any time and concurrent system activity, such as spooling of input and output, has negligible effect on user throughput. The entire computing facility is assigned to a user for the duration of his run (job) - a logical entity defined by the user by means of system control cards which consists of a sequence of program executions and system service functions. The charge for a run is proportional to the length of time it is on the computer. The fairness of this procedure is based on the following considerations:

1. Identical runs have equal running times - measured from the time of day that a run begins until it ends.

2. It is impossible to use part of the system without preempting all of it. This makes it fair to charge in proportion to time on the machine without considering which facilities a run is actually using.

Both of the above are no longer true to multiprogrammed systems. Since several runs may be processed concurrently, the interval between the starting and ending times of a run varies with the mix in which it is processed. Furthermore, since any facility not being used currently by a run is theoretically available to other runs, each user feels that he should be charged only for precisely those facilities he uses and only for the time that he is using them. This feeling is reinforced by the fact that whereas the capacity (and hence the cost) of a batch system is designed to meet the needs of the largest problem one expects to run on the machine, the capacity of multiprogrammed systems - because of the expectation of concurrent operations - tends to exceed by far the needs of any individual run.

Therefore, it has become necessary for multiprogrammed systems to keep more detailed records of the activity of each run. It is no longer possible to record that a user got on the machine at 3:15 and off at 3:23. It is necessary to record for what part of that interval the run actually had control of the CPU, how many I/O operations it performed, how much memory it used, etc. The level of detail will depend on a trade-off between the cost of accumulating the data and the difference it makes in charges passed on to the user. For example, suppose that the size of data records read from tape varies between 5 and 10,000 words with a mean of 250 words and a small standard deviation. Under those conditions, most users won't object to being charged for 250 words of channel time for every tape I/O operation. So unless the cost of gathering more detailed data is very small, it does not pay to record the actual number of data words transferred for each I/O operation. In considering the cost of the data recording, the cost for the programming effort will in general be greater than the machine cost incurred by the execution of the data gathering code over the entire life of the system.

While cost accounting has provided the incentive for the inclusion of detailed recording (logging) capabilities in the design of multiprogrammed systems, the information once gathered can be used to evaluate the technical performance and efficiency of the system. Furthermore, once the basic logging mechanism is available, the incremental cost of gathering technical (non-accounting type) information about system characteristics is very small. This behavioral picture of the system can be used to:

1. Isolate those parts of the software in which an increase in efficiency would yield the greatest return,

2. Spotlight goofs and inappropriate algorithms,

3. Suggest cost-effective hardware modifications or additions to the system.

II. Objectives

We would like to use the logging/charging system in the following ways:

1. Apportion the costs of running the computer facilities.

2. Control of programming errors. We will require the user to give a maximum to the amount of charge units his run is allowed to accumulate and terminate the run when this is exceeded.

3. Minimize the mean turnaround time for runs by giving high priority to runs with low charge estimates.

4. Help the user to program efficiently. A reduction in charge should be indicative of an increase in efficiency.

5. Use the charge/hour processed by the system as a measure of efficiency. While each individual user is trying to reduce the charge for his run, the system should attempt to maximize its charge capacity.

6. Control of programming practices. By charging more for overloaded facilities we hope to achieve a more balanced load on the system. Since computer charges are formal rather than real in our environment, we hope to achieve this by offering higher priorities to users with low-charge jobs.

In order to attain these goals the logging system and the charge generated by it must truly reflect the cost and impact a run is making on system throughput. Every decrease in charge by a run should represent an increase in efficiency and programmers should be urged to take all reasonable measures to reduce the charge of their runs. The charge must be computed dynamically instead of at run termination in order to attain goals (2) and (3). If the priority algorithm is to be meaningful, (2) is a prerequisite for (3). The charge for a run must be the same regardless of the other activities of the system at the time. This is necessary both for psychological reasons as well as to attain goals (2), (3), and (4).

Having the charge reflect true cost and keeping it invariant for a given run are to some extent conflicting objectives. For example, if a user does I/O on a saturated channel, the request takes longer and hence the user's core is tied up for a longer time than if the I/O had taken place on a little used channel. Hence, we should charge more for use of saturated channels. However, this charge cannot vary with the condition of the channel at the time the request was made but must be based upon the statistical characteristics of channel use.

The important part of this effort is the logging. Once this has been done, the method used to generate the charge can be varied according to installation objectives. There is in fact no reason why several different charges cannot be generated. For example, C1 could be used to drive the priority algorithm, C2 for billing purposes and C3 as a measure of system throughput. In order to provide a proper base, the following information must be logged:

1. The amount of CPU time used by each program. This should include the CPU time used by the system in processing user requests.

2. The channel time used by the program. For many devices, this time will vary with the state of the device when the request is submitted. In order to keep the charge to the user invariant, we must log the average time that this request will keep the channel busy. In order to do this properly, it is necessary to log the number of words processed by the channel.

3. The core-time product used by the run. In a multiprogrammed system, a run is often in core but doing nothing because the system is busy processing other runs. Hence, it is not sufficient to simply charge a user for elapsed time in core. Instead, we must keep careful track of when he is actually using his core space. This will be the case if he is either using the CPU or if a channel is actively transferring data into or out of his core area.

The information logged should be invariant for a given run. In some cases this is not possible. For example, the only way to measure CPU time is by means of the real-time clock which has a granularity of 1/5000 of a second. The amount of over-charge (or under-charge depending upon the algorithm) caused by this granularity depends upon system activity. Furthermore, the number of cycles actually available to a run for the duration of the time it is logically in control of the CPU is a function of the rate of I/O transfers. In addition, the number of cycles used by an instruction is one more if the data is in the same memory bank as the instruction than if it is in a different one. The difference in CPU time can be as great as 30% for typical programs. Whether this is the case or not depends upon what other programs have to fit in core concurrently with this one.

In deciding how much channel time to charge to a run for I/O activity, it is important to consider not only the device on which a user file resides but also the manner in which it was placed there. Suppose that a system has both high speed FH432 drums and the slow FASTRANDs. Suppose the user allowed the system to allocate the file to whichever device was free and the system allocated his file space on the FH432. It would be wrong to charge him the lesser channel time for the high speed drums because the next time this run is made it may be necessary to place the file on FASTRAND. Instead, the system must consider the probabilities of the file being placed on each device and charge a weighted average of the actual channel time for each device as the channel time for that file. This is particularly crucial when it comes to computing the core-time product. On the other hand, if the file was placed on a device because the run specifically required that this be done (if space was not available the run would wait until it was) then it should be charged the channel time appropriate to that device.

In general, the core-time product logged against a run for I/O activity will depend upon the channel time needed by the request. However, it need not be restricted to this. The statistical characteristics of the channel should be taken into account. The average wait time in the queue should be added to the channel time in figuring the core-time product. This must be done independently of the state of the queue at the time the request is made since the core-time logged against a run must remain invariant.

In computing the core-time product, it is important to keep track of concurrent operations. In the above discussions, we have assumed that CPU and I/O generate an interval during which the run has effective use of core and that it is appropriate to charge for the use of core during that interval. However, the core charge should be no greater if two or more such activities are taking place simultaneously. Hence, if any of these intervals overlap (i.e., the operations are concurrent) it is important that we use their union rather than their sum in the core-time product for otherwise we would be charging twice for the same core use. Without this feature in the logging system, the user has no way of knowing the effect of multiprogramming within his own run.

In considering the charge to be made for the use of the different facilities, it is important to remember that the effects of a run are not restricted to the facilities actually used. By its very presence in the system, it has preemptive effects. In batch systems, this was obvious. Each run had total control of the system. In a multiprogrammed system, it is a lot harder to determine just what this effect is. For example, the probability of the CPU being idle increases as the number of runs in core decreases. For each run, the amount of core it uses decreases the probability of their being enough runs in core to use the CPU and hence that portion of CPU idle time ought to be assessed to that run. The limiting case is reached when one run uses all the available core. In this case, it should be charged for all CPU idle time just as it would have been in the batch system. The same considerations apply to idle channel time. So the cost of core-time use should be based not only upon the cost of core, but also on its effect on the probability of being able to utilize the other facilities.

The extent to which a run inhibits the multiprogramming of other runs depends on the environment. In a system where two runs saturate the I/O channels, the size of a program is not very important. If the system is compute-bound, then I/O-bound runs have very little impact if sufficient memory is available. The converse is also true.

III. The UNIVAC System

The EXEC 8 operating system delivered by UNIVAC for the 1108 does a very poor job of gathering information for the purposes discussed above. At the current time, EXEC 8 keeps track only of the central processing unit (CPU) time used by each job. There is no logging or control over I/O activity or core size. It is possible for a large program in an I/O loop (due to program error) to dominate the system for hours and drastically reduce efficiency. Furthermore, it has long been the practice at Bellcomm to reduce the mean turnaround time by giving higher priority to runs with shorter running times. However, in a multiprogrammed system, the effect a program has on throughput and hence on the mean turnaround time depends more upon its size and elapsed time (measured when running by itself) than the CPU time. This is especially true at Bellcomm where the mix of programs is I/O bound so that there is very little correlation between CPU and elapsed time. This makes it unreasonable to apportion computer costs and futile to assign priorities as a function of CPU time.

IV. The Bellcomm Implementation

Because of these problems, we have modified EXEC 8 to log core-time and I/O operations as well as CPU time. Core-time is a product of the amount of core a run is currently using and the amount of time for which it has effective use of it. The decision as to what use is effective is made within the design constraints of the system. For example, suppose that a program that occupies 30,000 words of core wants to write a block of 250 words to tape. It informs the system of this need and defines the tape file and location of the data to the executive system. At this point the executive takes control, performs the operations, and gives control back to the program when the I/O operation is complete. Now theoretically the program needs not all 30,000 but only 250 words of core while the operation is taking place. However, the system design is such that the program must remain in core while I/O is taking place. Furthermore, even using the high speed drums it takes longer to swap out the unneeded 29,750 words of core and load another program than it does to write the 250 words of data to tape so that it is physically impossible for any other run to use this memory space while the I/O is taking place. Hence, the use of all 30,000 words of core is considered effective for the duration of the I/O operation. Suppose, however, that the I/O operation cannot be performed immediately because some other run is currently using that channel. The time that

the request waits in the I/O service queue does not constitute an effective use of the program's core space since it would not have occurred had the program been executing alone.

The units of core-time are rather arbitrary. In the current implementation, one core-second is the use of 8192 words for one second. The use of core-time is incremented in units of 1/5000 of a core-second. CPU time is measured by the system in increments of 1/5000 of a second so the effective core-time use due to CPU activity can be calculated easily. It is much harder to determine the effective elapsed time to be charged for an I/O operation since this varies with:

1. the device on which the file resides,

2. the number of words being processed,

3. the state of the device at the time the operation is initiated.

In case (1), it is desirable to charge different rates only in those instances where the user has control over file residence. For example, a tape file as opposed to a drum file. If the file is a catalogued file, he should not be charged less because the system fortuitously allocated him space on the fast FH432 instead of the slower FASTRAND drums. On the other hand, a user should receive credit for assigning scratch files to specially reserved FH432 areas (see the drum @ASG Statement, Section 5.5.1.5 of the EXEC 8 Programmer Reference Manual, UNIVAC publication UP-4144 Rev. 1).

In case (2), it is obvious that a user should be charged more for larger data transfers. The precise relationship depends upon the device. In the case of tape, one can calculate the number of inches needed for a given number of words, add on the size of the interrecord gap and charge on that basis In the case of drum I/O there is a large initial cost per record and then a small increase in cost per word. For example, a record of 400 words written on tape takes about 5/3 as long as one of 200 words, whereas on FASTRAND the ratio is about 1.43.

Case (3) should not be reflected in charging effective elapsed time to the user since the state of the I/O devices (e.g., tape moving or not, positio. of FASTRAND boom) depends to a large extent on the mix of jobs being multiprogrammed.

In addition to the elapsed time charge for core during an I/O operation, there is also a charge for the use of the I/O channels and device controllers. Theoretically, this should also vary with the number of words being processed and the device being used.

In the actual implementation we came nowhere near following these recommendations. The reasons for this are that:

1. It was important to quickly get some accounting on the system that was not totally restricted to CPU time so we made some shortcuts for speed of implementation.

2. We do not know enough about EXEC 8 to implement some of these features at this time.

3. In some cases, the game is not worth the candle. For example, almost all tape I/O is done in blocks of 224-256 words. If we charge each user on the basis of 250 words per tape operation nobody will complain and we will be sufficiently close to estimating the real cost so that it would not pay to measure the number of words actually transferred during a tape I/O operation. What we have done at the moment is to assess a run for 80 milliseconds of useful elapsed time for I/O operations. This is probably too little for FASTRAND I/O and too much for tape operations.

In addition to logging this information, a charge is computed for each run. Let C be the number of core-seconds (in the current implementation this is the use of 8192 words for one second) used by the run, T the CPU-seconds, and I the number of I/O operations. Then in the current implementation

$$\text{charge} \cong \frac{C}{60} + \frac{T}{30} + \frac{I}{1000}.$$

In addition to the core-time logged to a run for every I/O operation, the I/1000 represents a charge for use of the channels, controllers, tape drives, cycle-stealing, etc. This charge should vary with the channel used, the number of words

transferred, etc. However, a standard charge was chosen for just about the same reasons we discussed above for assessing a standard charge of 80 milliseconds effective core use for each I/O operation.

The charge is computed dynamically. For every increment to CPU-time, core-time, or I/O activity, a corresponding increment is made to the current charge. Charge is incremented in quanta of 1/300000 of a unit. CPU-time logging is done in quanta of 200 microseconds so that the minimum increment of core-time is for the use of 8192 words of core for 200 μs. The charge for this is 1 charge-quantum. The charge for the use of the CPU for 200 μs is 2 charge-quanta. The charge for each I/O operation is 325 quanta.

The maximum allowable charge for a run is specified in the time field (now the charge field) of the run card. This field was chosen for the following reasons:

1. An estimate of CPU use is of little importance to an installation such as ours which is predominantly core and I/O bound.

2. We wish to assign priority to a run in inverse proportion to the charge estimate. The system already contains modifications to do this using the time field.

3. The system already contains facilities for terminating a run when the current value associated with this field exceeds the estimate. This is a necessary feature for enforcement of the priority algorithm.

The reason for incrementing the charge in quanta of 1/300000 of a unit is that the time field was represented in this way. The real-time clock increments every 200 μs so that each minute of the time field is most naturally represented by 300,000 such increments.

At present charge is computed by

$$\alpha C+\beta T+\gamma I. \qquad (1)$$

This constitutes a charge for the dynamic facilities only. Since the FASTRANDs account for a substantial portion of the computer system cost, any bill presented to the user should

include charges for track-hours of file residence on FASTRAND. This charge is a function of file size and is in no way related to the dynamic charge for accessing the file nor to the number of times it was used. The rationale for apportioning priority as a function of charge is that giving priority to runs that demand least of the system is the best way to minimize mean turnaround time. This is true only for dynamic use of facilities. Static use, such as size of files, has no effect on turnaround and therefore should not be included in any charge parameter that is used as a priority distributor.

In Table I, the first line gives the approximate cost/month of the devices considered in the charge parameter. The second gives the approximate availability in terms of what is left to the user after the system takes what it needs. The third line is an estimate of the fraction of that facility which is actually used at Bellcomm. The fourth line-adjusted cost is the cost factor assigned to that device. The values of α, β, γ in equation (1) were adjusted so as to contribute to the total charge in the ratio of the fourth line of Table I.

	Core	CPU	I/O Devices
Cost/Month	$20K	$14K	$20K
Availability/Sec.	100K work-sec.	1 CPU-sec.	40 operations
Fraction Used	1.0	.50	.50
Adjusted Charge	40K	10K	10K

TABLE I

COST OF SYSTEM FACILITIES

The system is most efficient if all its facilities - core, CPU, I/O - are being used to full capacity. If a facility becomes saturated, we would like to provide an inducement to switch from that facility to another. However, we don't want them coming over in droves to tip the scale in the other direction. The adjusted charge line of Table I is our first attempt to provide this balance. Since we have two facilities - CPU, and I/O devices - which are not being used to capacity and hence not paying for themselves we must make up the cost somehow. This we did by raising the price of our saturated resource, core, to the point where we break even. At the same time, we are providing an inducement for our people to write smaller programs and use more CPU or I/O instead.

In assessing charges for good runs, it is important to remember that they are so only because of the environment. If we charge too little then, the users will tend to program so as to give their runs good characteristics. Eventually this will cause a change in the environment which may reverse the goodness characteristics. If everyone starts writing compute-bound programs because they are cheaper, the system will very quickly lose its I/O-bound characteristics. It is very important to strike a correct balance in assessing charges.

This is especially true if the charge estimate is used as a priority distributor. This tends to group low-charge runs together for multiprogramming. If all these runs are compute-bound then in effect, the system becomes compute-bound and the throughput will degrade accordingly. It is important that the mix of low-charge runs retains the system characteristics which made these runs candidates for small charges in the first place.

An alternative way to make up the cost would be to charge the unused CPU and I/O to overhead and distribute the cost to all the users. This would have the precise opposite effect of what we want. Users would tend to use as little of these facilities as possible so as to get the general overhead cost to pay for their runs.

V. System Throughput Evaluation

Let us suppose a fixed charge algorithm and a fixed set of runs are to be used as a throughput test. As we pointed out earlier, the charge to the user in a multiprogrammed environment ought to be completely independent of system activity. Hence, we

make the further assumption that for all the test runs the charge remains invariant with respect to system changes. Under these conditions it is obvious that any increase in charge/hour processed by the system does reflect an increase in throughput and conversely. It is immaterial whether this increase is due to software or hardware changes. This is true for biased as well as unbiased charge algorithms.

On the other hand, the cost effectiveness of a change (for sake of simplicity lets suppose its a hardware change) can be measured from the relative increase in charge/hour only if the charge is an unbiased reflection of the actual cost. In a biased system where we are overcharging for the use of critical facilities, an addition to those facilities will tend to show an apparent increase in charge/hour which is greater than the actual increase in the throughput.

We have been considering jobs processed as a measure of throughput. Suppose we reprogram the jobs in such a way that we can get the same answers by using less of the critical facilities. If the cost of the non-critical facility is sufficiently low, we can actually increase the throughput while decreasing the charge/hour processed by the system. In particular, for biased charge algorithms where the purpose of the bias is to change programming habits it is highly probable that this will be the case. It would seem that it is not possible to use the same measure both as an incentive to change and as a measure of efficiency. Moreover, the use of a single parameter to measure system efficiency does not really tell us enough about system behavior. It is much better to evaluate system performance with respect to a balanced use of all system components and use the charge parameter as a control.

VI. Recommendations for Further Development

The biased charge algorithm needs constant monitoring to insure that the bias is producing the desired effect. If the change in user habits is not sufficient, the bias must be increased whereas if the swing goes to far, we must reverse the bias. The following factors should be monitored in the current charge algorithm at Bellcomm:

1. The amount of elapsed core-time charge for an I/O operation. The current setting of 80 milliseconds may be too low.

2. The charge for CPU time may still be too high to achieve a proper balance.

3. We need to examine the effect on turnaround time of the priority categories. The number of categories should be reduced once the proper cutoff points are found to insure reasonable turnaround time.

4. The limits of the priority categories should be sufficiently large so that the jobs within a category contain a representative mix of I/O and compute-bound programs.

5. Is the charge for core-time high enough to induce the segmentation of large programs?

More fundamental changes are also necessary. The current algorithm which gives absolute priority to all jobs having a given priority letter over all others with a lower priority makes it very hard to achieve a proper balance and to give reasonable turnaround time to all categories. The behavior we would like to see from the system is that a job whose charge is 20 takes about twice the time to get done as one whose charge is 10. If the number of jobs in the 10-unit category is sufficiently large and is replenished at a steady rate, then it is possible that 10-unit runs get 1/2 hour turnaround while that for 20-unit runs is measured in hours. Besides being inequitable and undesirable in terms of efficient use of human resources (programmers in the 10-unit category are getting more and those in the 20-unit are less than they need to solve their problem), it induces the programmer to game the system and thus decrease both his own efficiency and that of the system. Someone running a 20-unit program may well spend hours breaking his run up into three 10-unit runs to do the same thing. And as long as we make it profitable for him to do so, that is not only what he will but also what he should do.

A proper scheduling algorithm will consider not only what priority category a run is in but also:

1. The length of time it was waiting in the queue.

2. The turnaround time it would have received had it been processed in a FIFO queue.

3. The amount of services that this user working on this problem has received recently. No one individual should be allowed to dominate the system, even if all his runs are small and therefore have high priority.

It will be necessary to reserve a few priority categories for jobs which are to have high priority and some categories at the low end for jobs which are not to be started until all others have been completed. This approach will also make it easier to achieve a properly balanced use of system facilities. For example, one could start one run from category G for every two runs in the system which are from categories D, E, or F.

The current charge for core increases linearly with the amount of core use. It is not at all clear that the effect on throughput is in fact linear. A careful examination of this question is necessary.

The current charge for I/O is the same regardless of the device or number of words being transferred. This is not a true reflection of the cost to the system. Not only are the FASTRANDs much slower than tape, but the FASTRAND channel is saturated so that use of catalogued files (assumed to be on FASTRAND) puts an additional burden on throughput which is not charged to the user since he is not charged for wait time in the I/O queues. The charge for tape and word addressable drum I/O should be appropriately reduced and that for catalogued files increased. This will yield a more equitable distribution of I/O costs as well as inducing users to move their files to tape or FH432 drum when it is reasonable to do so. In the current system, not only does the user have no incentive to change the residence of a file, he has no way of telling which is best since the charge remains the same regardless of what he does.

The same considerations apply to the measurement of concurrency. There is no logging of concurrency and hence the user gets no credit for overlapping of I/O and CPU processing or overlapping of I/O on two channels. Not only is there no inducement to try, but there is a way of telling how much is gained by doing so. If a user is processing two data files, he has no way of telling whether it pays to double buffer, put one file on tape and one on FASTRAND, or both on tape. A proper charge algorithm will help him decide.

I would like to suggest the following algorithm to accomplish this. Add a word (BCIOND) to the users PCT which is the time that the current I/O operation will end. Associate with each device - FASTRAND, tape, etc. - an elapsed time Td. Let To be the current value of BCIOND, Tc the current time, and E the elapsed core-time to be charged to the run. Then E = min (Td, max (0, Tc + Td - To)) and T_O = max (Tc + Td, To). That is, the remaining elapsed time of the current operation is subtracted from the elapsed time charge for this operation. If the result is non-positive, then there is no elapsed time charge for core use. The end time for the current I/O operation is then extended to include the time necessary to complete the operation just submitted. Similiarly, there is no core charge for any CPU time used while the value of To is greater than Tc. The changes necessary to include this algorithm (if you want to dignify it by that name) as well as to charge different rates for the different devices should be fairly easy to implement. While it may be difficult to charge on the basis of words actually transmitted, it should be fairly easy to vary the charge as a function of the number of words requested for transmission.

This system logs (and charges for) only some of the services provided by the system - namely the ER's to IO$, IOW$, IOI$, IOWI$, etc. It would be appropriate to include a charge (and perhaps logging) for every ER request the user makes to the system. This should cover at least the cost of processing the interrupt as well as any additional servicing time. For those ER's (such as TIME$) where the cost is the same for each request it can be calculated directly in the ER dispatcher (DISP). In those cases where the cost varies (such as PRINT$) with the amount of processing, appropriate changes will be necessary in the functions which process the ER's.

A. L. Rothstein

1032-ALR-dmu

Copy to
See Last Page