file

March 21, 1972

To: F. J. Corbató
R. J. Feiertag
R. A. Roach
J. H. Saltzer
V. L. Voydock
S. H. Webber

From: B. Greenberg

Subject: Proposal for a command metering facility.

The problem of command usage metering in Multics is analyzed, and alternate approaches to its solution considered. A general scheme to meter command use effectively is proposed.

Good idea

Can be installed as for Multics proper only. To test out?

How much time & working s.f. is needed to do "nothing" command? (What is how?)

To: Distribution

From: B. Greenberg

Date: March 20, 1972

Subject: Command Usage Metering Proposals

For several months I have investigated command metering in Multics, and have reaced some conclusions and developed some software to this end. In this memo, I will try to expound on what the problem is, what I have done to deal with it, and what I feel should be done further.

I propose to measure how many of the various Multics standard commands pass through the system each hour. I propose to measure the comparative popularity of these commands, by metering the usage of commands by command name. I propose the measurement of the average cpu time and generated page faults resulting from the use of each of these commands, and thus, the average number of page faults per average cpu second spent in these commands. As a longer range goal, I intend to measure the effectiveness of shared procedure by obtaining figures on the average number of users in a given command at a given time, and the resulting reduction in page faults thus created. These objectives should give us a better idea of where Multics is spending its time, and thus help us optimize its performance. Also, to help us validate our improvements, I propose to measure the total fraction of system up-time spent in these commands. This will help us judge the effectiveness of any optimizations resulting from figures derived from the meters described above.

Let us first define a 'command'. Ostensibly, a command should be any procedure described as such in the MPM or SPS. Considerations of implementability motivate us to limit our definition to those invocations of these procedures from the normal Multics command processor. As all the procedures described in the MPM or SPS reside in the system libraries, we can use this fact to differentiate between commands and user-written procedures. Hence, we define a command as 'any procedure residing in the Multics system libraries called by explicit request from the normal mechanism in the Multics standard command processor. This definition allows us a starting point for the construction of a metering tool, and gives us a clearer definition of what we are trying to meter. Note that this definition does not cover the calling of procedures classified as 'commands' in the MPM form within programs (not a recommended technique), but fully covers the use of exec_com, quit, active functions, and other similar recursive command use.

Secondly, we wish to define exactly what we wish to measure. We define the cpu time extent of a given command to be all the cpu time spent from the time

the Multics command processor calls the command procedure to the time the latter returns, minus the cpu time extent of any recursively invoked command. Note that this recursive definition always includes the system overhead of the actual recursion mechanism in the cpu time extent of the outer command in the case of recursive command use. This inaccuracy may be reduced by having the metering tools be knowledgeable about this overhead, and having them compensate for it in their meters.

Hence, we wish to measure, for each command as defined above, the total number of invocations and the sum of the cpu time extents spent therein, and thus the average time in each command by the division of these two numbers. We wish to count the number of page faults generated by a process during the extent of the concerned command, and add this to a system-wide total for this command. This should give us an idea of which commands have the smallest working sets, or make inefficient use of the working sets they have, e.g., the hypothesized problem involving the search of a list structure in the 'archive' command. With regard to the shared-procedure question raised above, we would like to know what percent of the cpu time and the page faults taken in a given command are accumulated while n invocations of that command are active, nbeing 1,2,3,4,'large number'. Although rather difficult to implement, the ring mechanism of the follow-on processor should facilitate the complex interprocess communication required to realize this aspect of the scheme. This last set of statistics appears to be a potentially powerful index for the measuring of procedure and core sharing. We can measure the total fraction of system up-time spent in these commands by accumulating all system time determined by the metering system not to be in the cpu time extent of any command, and using existant meters of cpu idle time.

There are several implications of these proposals which I feel need to be discussed.

The first problem is security. The entire command system operates in the user ring. Any temporary data which it accumulates is subject to willful destruction by a malicious user. Any special gates provided to attempt to provide some degree of security are user-callable by definition. Although intermediate measures can be taken to provide security against accidental destruction, it appears likely that there is no secure method of protection against a knowledgeable malicious user, in the current or proposed hardware. Among the measures which can be taken to insure against accidental or casual destruction of the master data base (to be discussed later) on a large scale is the rendering of responsibility for the maintenance of this data base to an administrative ring procedure, which will perform perfunctory gullibility tests on its arguments. However, when one realizes

that there can be no absolute security, and that the item whose security is at stake is metering data, not system integrity, a completely unprotected system in the user ring can well serve to provide us with the data we seek.

An implication of this security problem is where to put the master data base, containing system-wide accumulated command statistics. Since this data is capable of being destroyed, it should be freshly initialized regularly. Since anything which is initialized every time Multics is brought up, however, must be inspected regularly, if it contains metering data, it would seem that a permanent place in the heirarchy, say system_control_dir, would be an appropriate place for the globally-written data base. A copy accessible only to the priveleged should also be maintained, and used to compare to the master data base for report producing programs, as is standard with Multics metering programs . The resetting of the master data base should be done if and when the produced report indicates tampered data.

A second problem is the use of non-standard command processors. Since there is no way to tell how these 'command processors' interpret lines read in by the listener, and map them into Multics or other 'commands', there can be only two ways of metering command use generated by these alternate processors. On one hand, we may force commands to meter themselves by placing appropriate code in each command. Although this may be a valuable approach for special metering of a few specific commands (note that even this is subject to deliberate falsification), it is not at all general, gives us no handle on the total situation, and would require an inordinate amount of work to install in the existing command library. On the other hand, we could ask users who use their own processors to modify them to perform this metering. It is possible to modify the command utility segment cu_ to give us some idea of how many processes per day use their own command processors.

A third problem, related to the above, is the use of calls to commands by explicit call statements. Obviously, the non-general solution of putting metering in every command will handle this correctly. Other than this, it seems that no solution short of drastic would be adequate to obtain time and page fault data from such command use. Such a 'drastic' solution might involve reformatting linkages to procedures deemed to be 'commands' to force calls to metering procedures, this code being inserted by the linker when a link was snapped to such a procedure. However, since this technique is not widely used, and should not be, such a measure seems unnecessary. If there is concern over the use of this technique, a meter may be put in the linker to count calls (via the "link_fault" entry only) to procedures residing in the "sss" library.

A fourth problem is to distinguish between Multics commands and other procedures called from command level. The only sure-fire way of accomplishing

this is to ascertain whether or not an invoked procedure resides in the system libraries. In the current implementation of Multics, this requires a call into the hardcore ring. If this complete certainty is required, a new call entry point into hardcore should be defined, to return superior directory pathname along with the entry point pointer, when the command processor first searches for the desired procedure. Alternatively, I propose to assume that the incidence of user procedures (other than developmental stages of system commands) with the same names as library procedures (specifically 'commands') is sufficiently small as not to distort the resulting data. Thus, the name of a command can be taken as an indication as to whether or not it is a library procedure, simply by hashing it against a sufficiently large table of command names.

A fifth problem is the lack of efficient tools in the current implementation for the implementation of the sharing metering discussed above. For these statistics to be meaningful at all, we must count cpu time and page faults for every process using a command, and subdivide these figures as per how many invocations of that command were running for each second therein. Since the whole assumption of the value of these meters implies that the statistics are substantially affected by the number of invocations active, we must clearly peek into all processes running a given command at any time the number of invocations running changes. These inter-process data retrievals imply that the hardcore traffic controller data base either be read-accessible from the user ring, which seems unnacceptable to me, or from a gate which will extract the desired information. However, this gate would seem to have a potential for fairly heavy use, and it seems that the gate mechanism of the follow-on seems a first prerequisite for this scheme to be feasible in any way.

I have developed some software to create and maintain the data bases for a prototype system operating in the administrative and user rings. I have developed a prototype system, which runs in the user ring, and counts invocations of an arbitrary subset of Multics commands, and prints a report.

In light of all the preceding, I propose the following scheme: Provide a completely unprotected system-wide data base in the user ring, as described above. Provide, also in system_control_dir, a hash table, readable by all, writable by the priveleged, of all known command names and synonyms, regardless of whether it is desire to meter them or not. This allows more accurate determination of 'command' versus 'user procedure'. This hash table will map command names into "command indices", or into a reserved index if the name is not found. For each defined command index, have a three cell bucket counting invocations, totalled cpu time extents, and totalled page faults during those extents, all for the command concerned. Also provide a bucket for time and faults during time deemed not attributable to any command. This data base will be updated by things compiling to "asa" and "aos" instructions, and thus

need not be locked. The standard command processor will have two calls to a metering program inserted around its existing call to the sought command (currently implemented as a call to the command utility module cu_, entry cu_$gen_call). The first entry of the metering program, or "start", will serve to chain recursive invocations of the command processor, so that data for commands in progress may be updated. It will also record the process time (52-bit number returned by hcs_$usage_values) it is called, hash the invoked command name to get the command index, and obtain the process page-fault count up to this point. It will also clear out a counter for "cpu time extent accumulated, this command". All these quantities will reside in a storage structure in the current frame of the command processor. Furthermore, the "start" entry will increment the count of invocations of this command across the system by updating the master data base invocation count. Also, the "start" entry must check for previous invocations of the command processor, and compute cpu time extent and page fault figures for the previous command active, and add them to the master data base. The "stop" entry point (the one after the call to cu_$gen_call) will cause the final cpu time extent and page-fault figures for the command just terminated to be computed, and added to the master data base. Also, frames of previous invocations of the command processor are updated to indicate a new tentative starting time for cpu extent calculations, and similarly for page faults. In order to reduce the number of hardcore calls (this scheme implies at least two per command), the standard ready message printer could be rewritten to take advantage of the calls that were already made to compute these time extents. All time between "stop" calls and "start" calls on the same level have their data attributed to "no command", as described above. Suffice it to say that the proposed placement of process clocks in the user ring (and preferably, page-fault count accesibility as well) will help this scheme considerably.

A copy of the global data base will be kept, updated periodically, and reports produced from the difference between the two copies.

In summary, the scheme I have presented represents an effective compromise between security, system overhead, and the desire to obtain the desired data. It relies on the small incidence of malicious users, and system integrity is in no way compromised. Further compromises torward security (still compromises, not absolute security) may be made, but become efficient only on the follow-on. The scheme presented is simple, general, easy to implement, and extensible. In the current hardware, I believe it presents an effective menas of obtaining data on standard command use in Multics.