To: Distribution

From: Steve Webber

Subject: The Performance Problem

Date: 01/28/74

Introduction

This memo is being produced in order to spell out many of the performance problems that currently exist in Multics. It is meant as a statement of the problem and although the memo hopefully gives a plan for investigating possible improvements no concrete changes are now being proposed. The memo is intended to get the performance problem analysis under way with some initial directions.

The performance problem reveals itself in many ways. Probably the most striking is the effect that system load has on system throughput. Most other systems of which I am aware degrade response time when the load gets heavy but the throughput (i.e. the amount of useful work which gets done per unit time) remains about the same. The Multics behavior is quite different. With a full hardware configuration at M.I.T. and about 10 users (doing something) the system utilization is quite high. The "other" time in a "ttm" printout indicates 85%-90% of the available processor time is being spent doing useful work for the users. This compares favorably with most other systems of comparable capability. However, when the user load gets up to 30 or 40 the useful time drops to around 50% of the available processor time and 70 users is even worse. This loss of efficiency is accompanied by the marked degradation in response time that might have been expected. The Multics system loses on both counts. This decrease in efficiency is probably entirely due to virtual memory considerations, and although there will probably be some decrease in efficiency when the user load gets heavy, it need not be nearly as bad as it is.

Other ways that the performance problem appears are listed below. The list is, of course, not exhaustive but shows what kind of problems exist.

1) backup can take up to 25% of the system resources;

2) the "nothing" command takes 60 milliseconds (with no type ahead);

3) some hardcore entries take much more time to do what they must do than it would appear to be necessary;

4) first time use of many commands in a process is much longer than subsequent uses (due to linking and command-specific initialization);

5) the call/push/return sequence is much too costly;

6) performance can get so bad that a user can not even get logged in (problem in scheduler as well as rest of system);

There are several projects that we would like to undertake that would require a "fast process" or at least a "fast process exchanger." The current traffic controller makes such designs impractical.

Another problem which needs attention, although not directly related to performance, is the inequity and unpredictability of the accounting methods used. Memory units do not do what it was hoped they would and in fact under certain loads memory unit charges are completely unfair. This is brought up here because users (system programmers in particular) will likely want to have reliable measures of what to control when trying to program for maximum performance.

Goals

There are a few goals which might be worth mentioning as we proceed with the study. These goals are listed here both to remind us of what we are striving for, as well as to impress the point that much work of a substantial nature must be completed this year. This means real long-range options should not be considered as a solution to our current performace problem.

1) The system should be able to run small and large processes (i.e. processes that require either large amounts of storage or CPU time) efficiently.

2) A scheduler should be provided which

a) allows system administrative interfaces for controlling system load, user priority, response time, etc.

b) allows user visible priority changes (on validated request).

c) provides information to page control and administrative "load levelers".

3) The system throughput curve must not deteriorate.

4) Some commands (and their environments) must be highly efficient so that a cheap service can be provided (BASIC, edm, runoff, tty I/O, fast FORTRAN, etc).

5) The capacity of the system, in terms of throughput, should be increased by at least 50% by the end of 1974.

The rest of this memo is divided into four sections descibing in more detail what areas might well be investigated. These sections are:

1) the hardware constraints;

2) the virtual memory problems;

3) other software of interest; and

4) a list of areas for potential investigation.

The Available Hardware

The speed of the various hardware components on which we run Multics obviously plays an important role in the overall system performance. As it turns out, two of the components, the CPU and the bulk store, are not as fast as we expected. This should not have too much effect on the system software (the only part of the system we can control) except that a basic decision in the paging algorithm was based on a bulk store transfer rate considerably higher than we observe. We should carefully rethink this and related algorithms in light of the real speed of the bulk store. The other hardware features of immediate interest to a performance analysis is the CPU speed. The 6180 with its EIS capability is a strikingly different machine for which to generate code than the 645. In addition, there are several instructions and modifiers which are considerably slower on the 6180 in relation to code sequences which perform equivalent functions. Repeat and repeat double are slowed down considerably due to a crippling of the overlap features of the hardware. Direct modification (i.e., providing data for register loads without going to memory for the operand) is considerably <u>slower</u> than making the actual operand fetch from memory. Some code sequences can run up to 30%-40% faster if started on an even word rather than an odd word. These examples are meant to show that there are nonobvious ways to generate optimal code and although it could be a mistake to program such dependencies into a code generator it may be a valid target for investigation.

Other hardware constraints, such as disk channel capacity, do have a direct bearing on performance but our current software can make no meaningful decisions on changing such capacities. It would be interesting to know how best to change the scheduler and paging algorithms if, say, the disk channel capacity were halved (doubled).

The main features in the 6180 hardware that we did not have in the 645 are currently or will soon be exploited. The ring hardware is already being fully utilized while the EIS code generating compiler is only awaiting hardware reliability. The recompilation of the system to take advantage of EIS is an obvious and easy (but limited) way to improve performance. We should not depend on any performance gain here. We must find the flaws with the current system algorithms and strategies. Replacing code with faster code is nice but should not be considered a solution to our problems.

It may be noted that the PL/I compiler when recompiled with the EIS code generating compiler resulted in a compiler some 15%-20% faster (if a listing were generated).

It turns out that some performance problems of the CPU's are being investigated and that some improvements may result. We should not, however, depend on this either.

The Virtual Memory Problems

The virtual memory problem has two separate and for the most part independent components. These are 1) the software to implement the virtual memory and 2) the software that must run under the virtual memory. Nearly all code in Multics (including segment and directory control) falls into the second component. Page control, core control and the "file system DIMs" constitute the first component along with the necessary capability for process switching provided by the traffic controller.

One obvious area for investigation is page control and the algorithms it uses. For the first time in the Multics development I think we are forced to consider the scheduler and page control as a single entity with each giving more than token information to the other to aid in the execution of their respective algorithms. In addition to new possible algorithms that might result from such an organization we should also investigate how we can optimize those parts of page control that are independent of specific algorithms.

Some ideas that come to mind with respect to possible changes to paging are:

1) associating priority with core blocks

2) page stealing

3) new working set estimators

4) bulk store optimization

5) PC/TC integration

6) distributed locking strategies

7) demand prepaging

8) scheduling changes based on paging behavior

9) per-process page pools

Needless to say, other ideas have been mentioned and still more will arise if we perform an intense investigation into the paging behavior.

It is very likely that we will want to update and improve many of our metering techniques with respect to paging. We, of course, must have faith in what our meters say, as well as be convinced that they are accurate enough for us to be able to make judgements about <u>relative performance</u>. The entire metering issue should be reviewed at an early time so that we will have the tools ready when we need them.

The second component of the virtual memory problem (i.e., the success of programs running under the virtual memory) probably is the most important area to be investigated in that here is where the most progress can be made. Thrashing is more the fault of the system being paged than the paging system. There is nothing the most sophisticated paging algorithms can do about a program that references hundreds of pages in a very short period of time.

What is needed before we can proceed very far in this area, however, are again better tools and meters which give us information about where problem areas are and how the problems are manifested. Commands such as page_trace and sample_references are a first step and should be used where appropriate. Other commands such as meter_gate and link_meters give another dimension to the problem by showing how the virtual memory overhead actually affects system functions. The meter_gate program gives a clear indication where the time is spent in explicit calls to the supervisor. We should investigate each entry into ring 0 and optimize where appropriate.

Another important tool we will need is a consistent and representative test load that can be applied to new systems as changes are made. The assurance test we have now is a good starting point and it could probably easily be extended to meter any new functions we decide are appropriate. It does have the disadvantage that it requires a full or at least stand-alone configuration and hence is not as easy to use as on-line tests.

Since the main difficulty with running programs in our virtual memory is apparently the working set sizes we end up with, we should make a strong effort to minimize these where possible. The bind order is mentioned often in this context but this is a crutch which has limited effect. What is probably needed is a very detailed restructuring of code, internal procedures and the like to get code together which is really needed in the same time intervals. New tools to generate page reference strings and the like are needed and being prepared.

In contrast to paging code however, is the problem of paging data. The stacks, linkage sections, free areas, and KST are all commonly referenced data areas which get heavily paged. Minimizing the size of any of these can be of significant value. As a typical example the project of combining the lot and linkage section in the header of the stack should be revived and finished.

Other data areas such as the tree structure of the PL/I compiler have already been reorganized to minimize paging but more progress can probably be made.

Certain changes to directory control have been proposed that are worthy of study. Moving the lock out of the directory eliminates a page modification that page control can not know should be transparent. Also there have been structuring decisions made because of paging that may in fact cause more paging. An obvious example of this is the structure of an ACL. The actual personid and projectid names are not stored in each ACL but are shared by all ACLs in a directory. This does minimize disk usage but it also increases paging activity.

It has often been noted that backup could make great use of two new simple interfaces to the storage system. Although backup will be completely redone over the next two years it may be worth writing these interfaces today.

Another point worth mentioning is that of the <u>system</u> working set. Many changes in the system have been done with the argument that the change only adds a few hundred words (an occasional page fault!) to a user's address space. When there are 70 users on the system, 40 of which run in a 10 second period, however, the concept of a system working set should be recognized. Apparent small changes <u>do</u> have an affect on the whole system.

One last point of a general nature is that it might be worthwhile for more incentives to be available for good coding practices and the like. Currently there is little a programmer can do to generate a better program set no matter how hard he works. If the system rewarded those programs that exhibited those attributes most amenable to Multics, programmers could at least have a target to work for. They might have to sacrifice some to get programs to run well but if there were a set of rules the system recognized and a written description of these, there might be some hope.

Random Optimization

There are many areas of the system which need to be optimized and improved and which have nothing to do with the virtual memory. Poor code, unneeded generalizations, prosaic PL/I coding styles and remnants of EPL code exist in many areas of the system. The entire system should be reviewed with the purpose of cleaning up such code and replacing it with code for which the compiler can generate the best (EIS) object code. Coding rules have changed and these need to be written down and distributed.

Besides generating better code to perform the same functions, we should investigate if the functions are designed right to begin with. The following are typical questions of possible performance improvements:

1) should the name of the working directory be represented as a character string in the address space of each ring? (Do we want a working directory per ring?)

2) Same questions about the search rules.

3) Is the find_ mechanism of directory control acceptable?

4) Can ITP pointers really be used in argument lists?

5) Should the many components that get invoked in a command reference (command_processor_, abbrev, do, cl_edit, exec_com, etc.) be combined and better structured to permit fewer passes over the character strings, fewer stack frames, etc?

6) Should we design special systems with very small KSTs, linkage and stacks for fast BASIC processes?

7) Should we provide special system administrative tools to "wire" pages of or otherwise improve performance of selected subsystems (at the expense of others)?

The examples are typical and there are many more. We must merely get to work and check into everything for which we have time.

Current System Utilization

The following percentages (of real CPU time) give a broad indication of where our problem areas are. The percentages given are not exclusive (overlaps exist) and in general apply to a fairly heavy load.

Page Faults Percentage	%
Ring 0	40-45
Directories	12-15
Process Dir Segs	30-40
Level 2 (libraries)	25-30

System Overhead	%
tty DIM (not interrupts)	8-9
page faults	30-40
seg faults/bound faults	2-5
interrupts	
IMP	1-2
tty DIM	5-10
other	4-8
total ring 0 calls	40-45

Ratio of Virtual Time to Real CPU Time (efficiency?)	
Heavy Load	15-30%
Medium Load	50-60%
Light Load	90%

What we can do

The following is a list of study areas that would probably be worthwhile looking into. One purpose of this memo is to come up with a fairly complete list of such problem areas and the reader is therefore urged to contact me if he has items which belong on the list.

1) Study page control for any possible optimizations.

2) Rework algorithms in light of processor and bulkstore speeds.

3) Rework algorithms for page removal by integrating the traffic controller.

4) Redo the scheduler to take into account paging activity.

5) Page stealing.

6) New working set estimators.

7) Several page control locks.

8) Demand prepaging.

9) Update and improve all metering tools.

10) Find which new meters are needed and generate new tools.

11) Get a consistent, representative test load.

12) Rebind to minimize page faults - needs tools to help with decision.

13) Analyze stacks, linkage, etc. to shrink where possible.

14) Study the most commonly used commands and subroutines - analyze the algorithms and code used.

15) Study all supervisor entries for algorithms and code.

16) Determine if the system interfaces are inherently slow and inefficient.

17) Redo directory control to minimize paging.

18) Should we provide special interfaces for backup.

19) Should the system reward sharing more than it does now (sharing can help the overall system efficiency by minimizing the <u>system</u> working set).

20) Should a more efficient signalling mechanism be used - should "static handlers" be available for any faults/signals the user wants to trap.

21) Should the tty DIM be rewritten - or at least should we rewrite the call side interface routines (tty_write, tty_con, etc.).