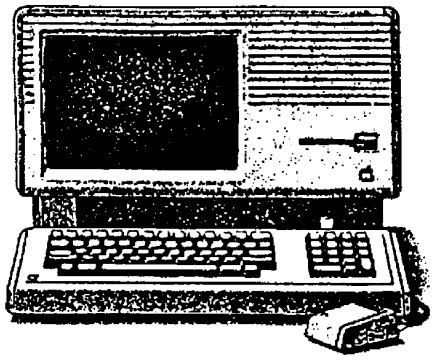
Document# 473

Apple Lisa Information



FILE NAME
A2 Performance:
Observations and Recommendations

DISK #

COMMENTS
18 Oct 1982

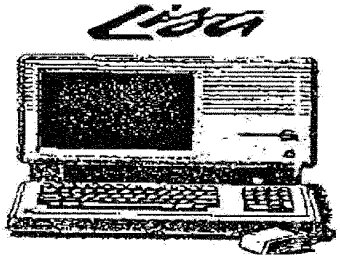
5 pages

David T. Craig
736 Edgewater, Wichita, Kansas 67230
(316) 733-0914

FROM: Mark Deppe DATE: 10/18/82

TO: Bruce Daniels
Frank Ludolph
Rod Perkins
Tom Malloy
Chris Moeller
Art Benjamin
Wendell Henry
Paul Williams
Wayne Rosing

CC: POS Software



RE: A2 Performance - Observations and Recommendations

For the last two weeks I have been using the OS logging utility and running the Logutil program to see what's going on inside our little box. I have noted down some observations and recommendations which I thought would be useful to pass along. All of the tests were run with 1 megabyte of memory, a profile, and using the A2 release, or preliminary forms of it.

Observations

- Swapping in a code segment usually takes between .5 and 2 seconds from a profile, and this is fairly proportional to the size of the segment. Very large segments tend to take about 2 seconds, and very small ones about .5 seconds. This includes the time spent by the memory manager moving things around in memory and discarding other segments.

- It requires about 41 swaps to pull a list manager document for the first time. This results in about 48 seconds during which the list manager is doing nothing but waiting for swaps. This also happens to be 56% of the overall time that it takes to pull a list for the first time.

- When the contents of memory were examined at a few snapshots, I found that 25-30% of it was occupied by data segments of one form or another (stacks, syslocs, private and shared data segments). This does not include the portion of memory occupied by the OS, only that portion used by the filer and applications. The amount of memory available to the Filer and the applications seemed to be about 600K, 160 of which was occupied by data segments.

- There is a tremendous amount of code necessary to run either the Filer or an application. When left to itself, a single process will manage to completely fill the available 600K. Fortunately, a lot of this is shared code and data which can be used by other processes.

- Once a process begins to execute, it will usually manage to completely displace the private code and data segments of the previous process within a fairly short time. This means that if a process has been executing for

awhile, a process switch will require all of the new processes code and data segments to be swapped in.

- Because of the above, process switches can be extremely expensive. The most common switches occur between the Filer and the applications. How expensive it is is proportional to the amount of activity which occurred since the previous process switch. If process A has been executing a long time and switches to B, this will take a long time because few of the data or code segments that B needs are in memory. On the other hand, if B had recently executed, then the switch will be relatively fast.

- A handshake protocol is even more expensive. This involves two process switches of the form Filer/Application/Filer, or Application/Filer/Application. An example of this type of handshake occurs after an application has opened a document and sends a DocOpened message to the Filer. The Filer executes and responds by sending an Activate to the application. The application then begins to execute again.

  One of the worst examples of this occurs when a document is pulled for the first time and the application must initialize itself in addition to opening the document. By the time the application has the document open, the Filer is nowhere to be seen in memory. Bringing it back in to do the handshake requires about 13 seconds. Then there is an additional 9 seconds required to swap back in the application's segments which were just thrown out by the Filer. The total cost of this protocol is 22 seconds. The specific example used in this case was LisaCalc, but I have also seen similar behavior using the LisaDraw, and I would assume it is also true for most of the applications.

  This type of protocol completely defeats the clock algorithm of the OS memory manager. After the second switch back to the application, the OS continues to throw out the application's code segments, while leaving the Filer's intact. This significantly reduces the amount of memory available to the application for two clock rotations, after which time the Filer's segments are thrown out. In the meantime, however, there is much thrashing between the application's segments.

- The percentage of time the system spends swapping, as opposed to doing useful work is significant and varies depending on the task. Using the list manager as a test case, I found that when pulling a document for the first time, it spends 44% of its time waiting for swaps during the one-time initialization, 93% waiting for Filer segments to swap back in, and 65% of the time waiting for swaps when actually opening the document.

  When a process must be completely swapped in to perform a relatively short task, the swap percentage becomes the major portion of the time. In the above example, swapping in the Filer to perform the OpenDoc/Activate handshake required about 8.4 seconds, 7.8 of which were spent waiting for data and code to be swapped in. This included 3 data segments, 4 private code segments, and amounted to 93% of the time that the Filer executed.

- Some of the swapping caused by an application can be eliminated. I experimented with the List Manager and was able to reduce the time required

to pull a document for the first time from 104 seconds to 84 seconds. The total number of swaps was reduced from 58 to 41. This is a 19% reduction in time and a 29% reduction in swaps. This was achieved by careful study of the swap logs and by using certain techniques, the more successful of which are described later. Not all of these techniques may be practical for everyone, but some of them are guaranteed to improve the performance somewhat.

Recommendations

It became obvious from this exercise that no single change to the system is going to increase the performance by a large amount. There are a number of possible changes which when taken together could make a difference. However, some of these may not be practical in the short term due to the level of effort required. It would probably be to our advantage for the Performance committee to determine which of these techniques are feasible and in what time frame.

1) Reduce the number of process switches which are required. The tremendous success of the 'picture' technique was due to the fact that it eliminated the need for a great many process switches. Now the only ones left which are troublesome are the Filer/Application switches.

   The most obvious way to do this would be to change the Filer/Application protocols to reduce the number of switches. For example, when a document is pulled, the filer knows that the document should be activated as soon as it is opened. If the Open/Opened/Activate protocol could be compressed into a single message from the Filer, two process switches could be eliminated. Similar techniques could be done with other such protocols. Depending on the operation, I believe that between 5 to 20 seconds could be saved for various scenarios. This advantage should be amplified on a system with smaller memory or swapping to twiggies.

   Since it is too late to completely revamp the protocols for first release, one strategy might be to phase in new streamlined protocols to the existing set, starting with the ones which would improve the performance the most. Applications could then begin to use the more efficient protocols as time permitted.

   It is also possible to reduce the overhead of performing the existing protocols if the handshake is done differently, although the performance benefit is not nearly as great. For example, when the List Manager gets a request to open a document, it immediately returns the 'DocOpened' response. This is an outright lie, but because the Filer is still in memory, the process switch takes less time. This is due to the fact that the overhead of doing a switch from process A to B is proportional to the amount of activity which occurred between the last switch from B to A. Switching back to the List Manager to activate the folder also takes very little time, for the same reason. The List Manager then opens the document, and the extra code needed to accomplish this can displace the Filer without any penalty. A similar technique can be done when closing a document. By comparing the list manager logs with other application logs,

it appeared as though the 'quick' Open and Close protocols executed about 8 to 14 seconds faster than the extended protocols.

This is not nearly as good as eliminating the switch entirely, but it reduces the overhead without changing the actual interface. This option should be investigated further to determine whether other such protocols may be similarly speeded up, and to verify that there are no hidden inherent problems. By far the preferable solution, however, is to implement new protocols which minimize the number of process switches.

2) We should optimize the Filer as much as possible. This process is at the center of most switches, so it makes sense to spend a great deal of effort here. Specifically, the amount of code which is necessary to execute a protocol should be reduced to a bare minimum. Currently 4 private segments, in addition to a stack, sysloc, and private heap, must be swapped in to perform the DocOpened/Activate protocol. Any reduction in this overhead would benefit the entire system.

3) Everyone needs to carefully segment their code to be as efficient as possible with respect to swapping. This is especially important for hot shared code.

4) Make the memory manager more intelligent. Many of the possible techniques have been mentioned before. For example, adding another memory state to a segment would help. Weighting segments differently depending on whether they were private, shared, code, or data might work. Favoring the segments of the currently active process I feel would be a big help, if it could be done.

5) Reduce the amount of memory occupied by data segments. One good way to do this is to occasionally shrink heapzones down and readjust the the data segment memory size. Furthermore, data segments should be unbound if they are not used for any length of time during the execution of a process.

   Stack sizes could potentially be reduced by a small amount by going through the code and cutting out the historical dead weight of unused variables.

6) Reduce the number of data segments in the system. Data segments used by an application should be combined into one if possible. It would also help if the stack and sysloc data segments of a process could be combined into one.

7) Reduce the amount of code in the system. By this I mean, rewrite certain portions of very hot shared code to be more compact. I am not suggesting that we all recode in assembly, although Bill Atkinson has certainly used this approach effectively. However, it might be useful to identify the hottest shared code in the system and trim it down where possible.

   In addition, all code probably contains historical dead weight which could be trimmed off after a little examination.

8) Applications should be optimized as much as possible to minimize the amount of swapping which is required. I have found that the logging

facility is excellent for this purpose. First of all, the swap log should be examined to determine what segments are being brought in for various operations such as initialization, opening a document, closing a document, activation, deactivation, and so on. Using the log utility it is fairly easy to spot segments being swapped in which don't quite make sense.

For example, 'cold' segments which are swapped in during routine operations are a good indication that the segmentation is not quite right, and a procedure is being called which should really be in a hot segment. It may also indicate a call to a procedure which is not absolutely necessary. The converse is also true. Sometimes a hot segment is brought in when the application is doing cold functions such as initialization, an indication that a cold procedure resides in the hot segment.

In other cases, segments may be swapped in which need not be swapped in at all. Some initialization which is currently done every time a document is opened can be defered until the relevant operation is requested. The printer arbitration is one such example. Some applications need not execute this code until printing is actually requested. It saves swapping in a fairly large (and therefore expensive) code segment while opening a document.

During initialization, order can also be important. Sometimes segments are swapped in twice because the two references to it are separated by a large amount of other activity. By resequencing the two operations to occur together, one of the swaps can be eliminated.

Another trick during initialization is to initialize shared code first, and application specific code last in the sequence. When an application first begins to execute, there is a greater amount of shared code in memory then later in the initialization sequence, when some of the shared code has been replaced by the application's private code and data segments. Initializing code which is shared first means that there is a greater probability that the code segments needed are already in memory. Waiting until later often results in the segments being discarded, which then have to be swapped back in, potentially displacing valuable resident code. OpenScrap is an example of such an operation.

If possible, applications should defer creating data segments during initialization until the very last possible moment. This results in greater code swapping space during the initialization. If this cannot be done, it is sometimes possible to defer expanding the data segment until later, which achieves the same thing.

Shrinking the memory size of data segments at deactivation time, and any other appropriate time also increases the amount of memory for swapping. In addition, it will take less time to swap out the data segment when the process is not active, and take less time to swap it back in when it is reactivated.

All initialization which may be done only once should be done when the process is created, rather than each time a document is opened.

o The End o