The double paging anomaly

by ROBERT P. GOLDBERG
Harvard University
Cambridge, Massachusetts

and

Honeywell Information Systems
Waltham, Massachusetts

and

ROBERT HASSINGER
Liberty Mutual Insurance Company
Hopkinton, Massachusetts

INTRODUCTION

Belady's paging anomaly[1] has illustrated that certain page replacement algorithms can cause more page faults as the size of memory increases. Mattson[2] has shown that there exists a class of algorithms called stack algorithms (such as LRU least recently used) which cannot cause more page faults as memory size increases. In this paper, we investigate the dynamics of double-paging, i.e., running a paged operating system, e.g., IBM's OS/VS2,[3] under a paged virtual machine monitor, e.g., VM/370.[4] In particular, we show that an increase in the size of the memory of the virtual machine without a corresponding increase in its real memory size can lead to a significant increase in the amount of paging, even for the LRU algorithm.

DOUBLE PAGING

Virtual machine systems[4–8] provide the environment in which double paging phenomena can occur.* The core of a virtual machine system is the virtual machine monitor or VMM. The VMM has certain similarities to conventional operating systems in that it supports a user interface or "extended machine" on which user programs can be run. However, the extended machine supported by a VMM is the full functional counterpart of an existing computer, and may be the same computer that the VMM is itself running on. Since the VMM can support programs which utilize the full functionality of a computer system, the VMM can support complete operating systems in the same way that operating systems support user programs. It is thus possible to run two incompatible operating systems on a single computer at the same time, or to run one operating system in production mode while system programmers are simultaneously modifying another active copy of that same operating system.

Figure 1 uses IBM's VM/370 to illustrate the organization of a virtual machine svstem. The VM/370 Control Program (the VMM) is shown, running on a bare System/370 and supporting two virtual machines, VM1 and VM2. Since the virtual machines are identically equivalent to complete System/370 computer systems, any of the System/370 operating systems, such as OS/VS2 can be run on virtual machine VM1. OS/VS2, in turn, supports an "OS/VS2 Extended Machine" on which a user program is being run.

While paging is not an essential requirement for a virtual machine system,[5,7,9] the two facilities form a very powerful combination together. In particular, with paging, it becomes possible for the memory of the virtual machine(s) created to be larger than the real memorv. This facility has been used very effectively by CP-67 and VM/370 to run various operating systems and applications which require a very large memory.[4,10,11]

If the VMM supports a virtual machine which includes paging, then it is possible to run any paged operating system (including the VMM) on this virtual machine. In this case, we define:

- Level 2 memory—virtual memory of virtual machine
- Level 1 memory—memory of virtual machine
- Level 0 memory—memory of real machine, i.e., real memory.

The Level 2 memory is mapped via paging into Level 1 memory. The Level 1 memory is mapped, in turn, into Level 0 memory.* Under these circumstances. we have *double paging.*

Conventional computer systems do not provide direct

* In this paper we use the term "double paging" to refer only to these phenomena. It does not refer to any other popular uses of this term.

* The term *level* is used here informally rather than in the strict sense as defined in Goldberg.[6,7]

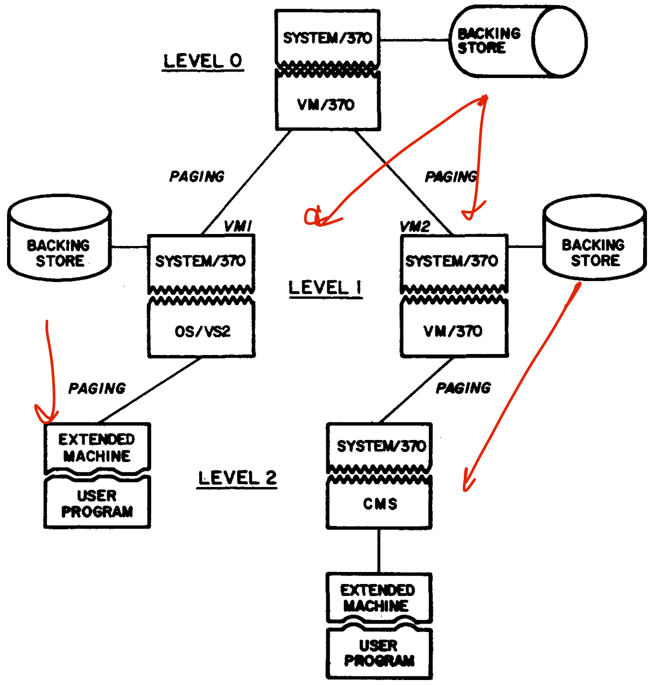

Figure 1—VM/370 virtual machine organization illustrating double paging

hardware support for double paging. Thus, in systems such as VM/370, software must be used to simulate the double paging *mechanism*. When a process is to be activated in Level 2 memory, its page mappings from Level 2 to Level 1 and from Level 1 to Level 0 must be combined to yield a single composed mapping directly from Level 2 to Level 0. Goldberg[7,12] and Parmelee[11] discuss the *mechanisms* for software support of double paging in conventional computer systems and Goldberg[6,7] discusses proposed computer architectures, called *virtualizable architectures* which typically provide direct hardware support for efficient double paging mechanisms. In this paper we shall not examine mechanisms. Rather we shall be concerned only with *policies*, i.e., the page replacement algorithms, and see how thev are affected by double paging.

Figure 1 illustrates two examples of double paging which can arise in the operation of VM/370. VM/370 uses paging to create the (illusion of) memory for virtual machines VM1 and VM2. OS/VS2 which is running on VM1, in turn, uses paging to create a large address space (virtual storage) extended machine for its user programs. On VM2, another copy of VM/370 is running, producing a second level of virtual machines. In the figure, the second level virtual machine is running CMS, an operating system which does not utilize the paging mechanism. Thus, both user programs shown in the figure will be affected by double paging.

Every paged system requires a backing store to preserve each page's contents when the page is not in the real or virtual memory. In Figure 1, the backing store shown at Level 0 holds pages of Level 1 (VM1 or VM2) which are not resident in Level 0 memory. The two Level 1 backing stores, in turn, hold pages of the Level 2 memory which are not resident in Level 1 memory.

The illustration of Figure 1 does occur in real world situations. As paging becomes more common in "target" machines, double paging will become more common in encapsulated systems. Examples of encapsulated systems include both the virtual machine systems (as illustrated above) for identical host and virtual machines, and integrated emulators[13] for dissimilar (paged) machines. An example of the latter might be running the PDP-10 TENEX System under an integrated emulator under OS/VS2*.

Other environments where double paging can be expected are in the newly proposed complex virtualizable architectures.[6,7,14] Related considerations arise in the management of multi-level (three or more) memory systems.[15]

DYNAMICS OF DOUBLE PAGING

We will examine the effect of choice of page replacement algorithms and sizes of memory upon the dynamics of double paging. We restrict our attention to demand algorithms operating in fixed memory spaces. Thus we will study one virtual machine at a time and ignore other effects introduced by resource multiplexing among VMs. Furthermore, when we examine the paging behavior of a reference string, we examine the behavior of the original string. We ignore any effective "rewriting" (or renaming) of the string which might occur.

Thus:

(1) We ignore "interference" to the reference string caused by any VMM traps and simulation.
(2) We assume that pages are treated homogeneously by algorithms, i.e., pages are not "locked."
(3) We assume page replacement algorithms and tables are not themselves in virtual memories. We assume they are external to the system or in hardware.

Loosening these ground rules makes the analysis more complex and might become the basis for future studies.

Furthermore, we make the following assumptions:

(1) Memory sizes in number of page frames are called n_0, n_1, n_2.
(2) n_0, n_2 with $n_2 \geq n_0$ will be known and fixed. page sharing.
(3) We can set n_1 but once set it will be fixed.
(4) The same algorithm is used for level $2 \rightarrow 1$ and level $1 \rightarrow 0$ paging.
(5) A demand page replacement algorithm is used and all free pages will be utilized, i.e., $n_1 \geq n_0$, $n_2 \geq n_1$.
(6) We count total page faults and ignore the fact that different backing store devices might be used at each level.

With the above assumptions and terminology we can

* This is merely a hypothetical example.

identify four distinct operating regions. They are illustrated in Figure 2.

(a) $n_0 = n_1 = n_2$
This case arises if $n_0 = n_2$. Then by assumption 5 (above) n_1 must equal n_0 and n_2. In this case, after the initial pages are brought into each memory no additional paging occurs.

(b) $n_0 = n_1 < n_2$
This case arises if $n_0 < n_2$ and we choose to set $n_1 = n_0$. In this case, after the initial pages are brought into level 0 memory there is only paging for level $2 \rightarrow 1$.

(c) $n_0 < n_1 = n_2$
This case arises if $n_0 < n_2$ and we choose to set $n_1 = n_2$. In this case, after the initial pages are brought into level 1 memory there is only paging for level $1 \rightarrow 0$.

(d) $n_0 < n_1 < n_2$
This case arises if $n_0 < n_2 - 1$ and we choose some intermediate value for n_1. Paging activity occurs for level $2 \rightarrow 1$ and level $1 \rightarrow 0$.

Case (a) is trivial and uninteresting. After start up, cases (b) and (c) exhibit identical behavior to a one level paging system. However, these cases remain of interest for comparison with the double paging case (d).

LRU ALGORITHM APPLIED TO DOUBLE PAGING

As noted above, stack algorithms such as LRU (least recently used) have a number of desirable properties. Among these is the stack property that any increase in memory size cannot cause an increase in the number of page faults.[2]

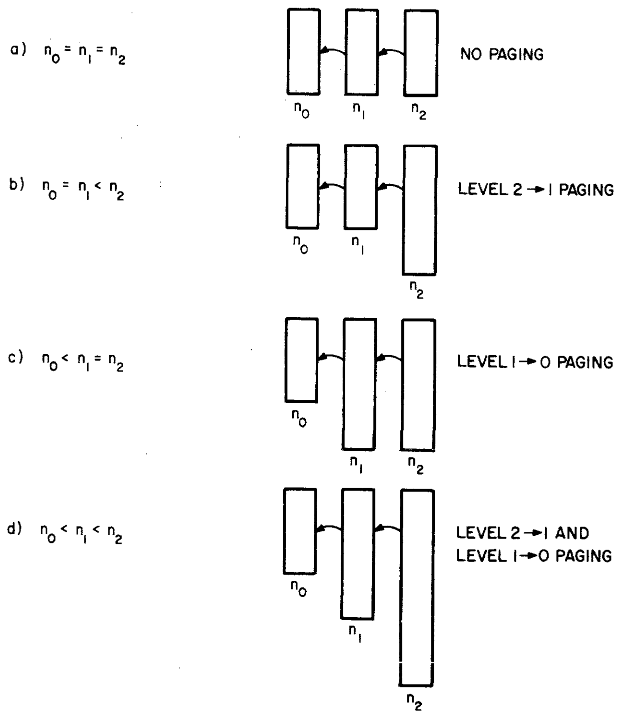

Figure 2—Dynamics of double paging

LRU 2 pages $n_0 = 2$ $n_1 = 5$	1	2	3	2	4	2	5	2	1	4	1	5	Reference String	
	*	*	*		*		*		*	*		*	Faults	4 Faults
	1	2	3	2	4	2	5	2	1	4	1	5	Stack	
		1	2	3	2	4	2	5	2	1	4	1		
LRU 3 pages $n_0 = 3$ $n_1 = 5$	*	*	*		*		*		*	*		*	Faults	
	1	2	3	2	4	2	5	2	1	4	1	5	Stack	4 Faults
		1	2	3	2	4	2	5	2	1	4	1		
			1	1	3	3	4	4	5	2	2	4		
LRU 4 pages $n_0 = 4$ $n_1 = 5$	*	*	*		*		*		*				Faults	
	1	2	3	2	4	2	5	2	1	4	1	5		
		1	2	3	2	4	2	5	2	1	4	1	Stack	2 Faults
			1	1	3	3	4	4	5	2	2	4		
					1	1	3	3	4	5	5	2		

Figure 3—Single level LRU: Increase in memory size cannot increase number of faults

In Figure 3, we illustrate the stack property by applying the LRU algorithm to a reference string which is run in a (conventional single paging) memory with three different sizes. The figure indicates page faults with an asterisk (*), and shows the stack contents at each point in time. In order to provide a uniform basis of comparison for all examples, faults are counted only after the largest stack has been filled.

The largest stack size is 4 and this stack will be filled after reference string elements 12324 have been run. We delimit this starting point with a dashed vertical line. and in Figures 3-6 count only those asterisks which fall to the right of the dashed vertical line. When the reference string consisting of five distinct page names ($n_1 = 5$) is run in a two page memory ($n_0 = 2$), four faults occur. When memory is increased to three pages ($n_0 = 3$), four faults still occur. When memory is increased to four pages ($n_0 = 4$), the number of faults drops to two.

In double paging, we must specify the reference string (same as above), the replacement algorithm (LRU), and the memory sizes (to be indicated below). Figures 4-6 apply LRU algorithms at both levels to the same reference string. We fix the reference string memory $n_2 = 5$ and the real memory $n_0 = 2$. Then, we count the number of faults as the virtual machine memory n_1 is varied.

Figure 4-6 are similar to the single level paging case of Figure 3. The reference string, stack values, faults, and beginning of count (dashed vertical line) are shown. However, in these figures, the TOTAL faults are given by the sum of level $2 \rightarrow 1$ faults and level $1 \rightarrow 0$ faults.

In Figure 4, reference string memory $n_2 = 5$, virtual machine memory $n_1 = 2$, and real memory $n_0 = 2$. After start-up we count 4 Level $2 \rightarrow 1$ faults and 0 Level $1 \rightarrow 0$ faults for a total of 4 faults. This is double paging case (b).

Figure 5 illustrates the anomalous double paging behavior which can arise in case (d). We keep the same real memory size $n_0 = 2$, and the same reference string memory $n_2 = 5$.

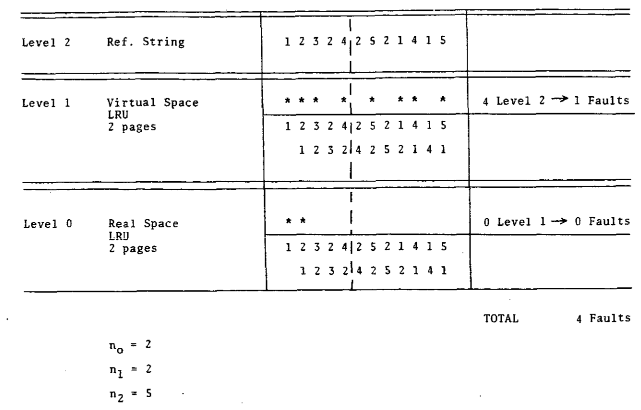

Figure 4—Double paging with LRU: Case (b)

However, we increase the size of the virtual machine's memory $n_1=3$. As can be seen, the paging behavior becomes significantly worse. There are still 4 Level 2→1 faults but now there are also 4 Level 1→0 faults for a total of 8 faults. Thus, the number of faults has doubled. Figure 5 details how this increase has occurred. The small circles in Figure 5 indicate those pages in Level 1 which cause an extra induced fault in Level 0. The arrows point from the Level 1 pages to the extra Level 0 pages which are needed. Finally, the large circles in Level 0 indicate page renaming operations, i.e. what used to be page 1 is now page 4. For the LRU algorithm, each Level 2→1 fault causes an extra Level 1→0 fault to occur also.

In Figure 6, we further increase virtual machine memory size $n_1=4$, keeping reference string memory $n_2=5$ and real memory $n_0=2$. There are still 4 Level 1→0 faults but the number of Level 2→1 faults has dropped to 2. Thus, the page fault total has decreased to 6.

We can summarize the results of this example as:

For a given reference string in a double paging system, an increase in the size of the memory of the virtual machine without a corresponding increase in its real memory size can lead to a significant increase in the number of page faults, even for the LRU algorithm.

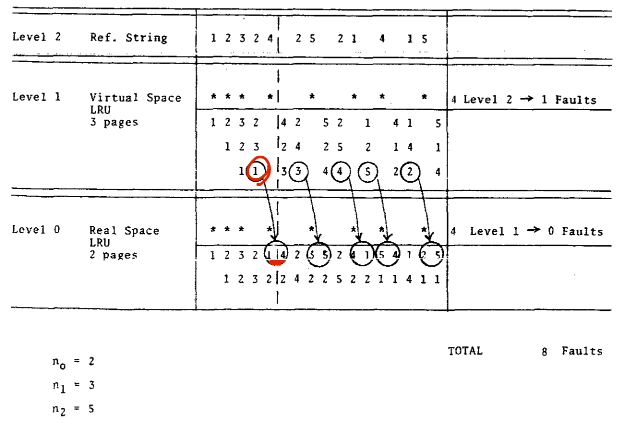

Figure 5—Double paging with LRU: Case (d)

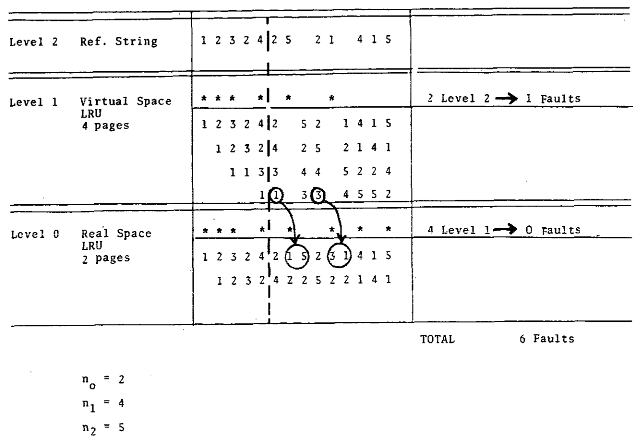

Figure 6—Double paging with LRU: Case (d)

THE ANOMALY EXPLAINED

The double paging anomaly occurs when $n_2>n_0+1$, n_0 and n_2 are fixed and we have $n_1=n_0$ (Case (b)). We increase $n_1=n_0+1$ (Case (d)) and the amount of paging increases significantly.

The anomaly can be explained for the LRU algorithm by an inspection of the reference stacks of Figure 5. The level 1 stack contains three entries whereas the Level 0 stack contains only two entries. When a page is at the bottom of the level 1 stack and is a candidate for next removal, it already has fallen out of the level 0 stack and has been swapped. Thus, in order to swap an entry from the level 1 stack, it must first be swapped into the level 0 stack.* Since LRU algorithms are operating at both levels, the level 0 removal algorithm will make exactly the worst choice each time.

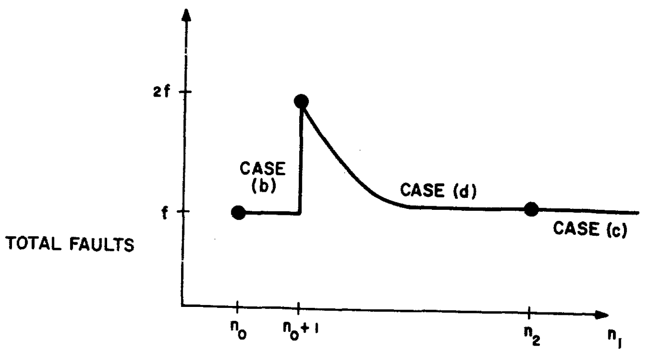

Figure 7—Page fault dependence on n_1 for LRU algorithm

* When an entry is "swapped" into a stack, its corresponding page is swapped into memory.

Thus, it will always have to swap in the page which it most recently swapped out.

For the LRU algorithm, worst performance occurs for Case (d) with $n_1=n_0+1$. As n_1 increases, the number of page faults continues to decrease (the stack property) until $n_1=n_2$ and we have Case (c). While the actual performance depends upon the reference string and memory sizes, the trend can be seen in Figure 7. The figure is drawn as a continuous curve even though it is really a series of steps.

Surprising results hold not only for the LRU algorithm but for other double paging replacement algorithms as well. For example, with the same reference string used above, the FIFO (first in first out) algorithm also doubles from 5 to 10 the number of page faults in going from $n_1=2$ to $n_1=3$. On the other hand, with the very unlikely MRU (most recently used) algorithm, the number of page faults remains constant as n_1 is varied between $n_1=2$ and $n_1=5$.

VM/370 avoids some of the difficulties explored in this paper through the use of certain specialized algorithms which allow locking (dedicated) pages in main memory. While this procedure may decrease susceptibility to the double paging anomaly, it reduces resources available to other users and might adversely affect global performance. In any case, we have shown that in double paging situations great care must be exercised.

As noted above, virtual machine recursion[5,7] implies the ability to run a VMM under a VMM under a VMM .... It is known that in order to test VM/370 software before System/370 hardware was available, IBM ran specially modified versions of CP-67 several levels deep. In the new complex virtualizable architectures *mechanisms* are provided for supporting arbitrarily deep recursion. In these systems, the double paging problem generalizes to the m-level paging problem.

CONCLUSION

Progress in understanding complex phenomena is often made through the discovery and explanation of anomalous behavior which arises in apparently simple situations. In this paper we have examined one aspect of the resource allocation problem in a large computer system. We have observed that when rational locally optimal algorithms are combined together, distinctly suboptimal global behavior can sometimes result.

REFERENCES

1. Belady, V. A., "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, Vol. 5, No. 2, 1966.
2. Mattson, R. L., J. Gecsei, D. R. Slutz and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, Vol. 9, No. 2, 1970.
3. IBM, *Introduction to OS/VS2 Release 2*, IBM Corporation Publication No. GC28-0061.
4. *IBM Virtual Machine Facility/370—Planning Guide*, IBM Corporation, Publication No. GC20-1801-0, 1972.
5. Buzen, J. P., U. O. Gagliardi, "The Evolution of Virtual Machine Architecture," *Proceedings AFIPS National Computer Conference*, 1973.
6. Goldberg, R. P., "Architecture of Virtual Machines," *Proceedings AFIPS National Computer Conference*, 1973.
7. Goldberg, R. P., *Architectural Principles for Virtual Computer Systems*, Ph.D. Thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, Massachusetts, 1972.
8. Goldberg, R. P. (ed.), *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, Cambridge, Massachusetts, 1973.
9. Goldberg, R. P., "Virtual Machines: Semantics and Examples," *Proceedings IEEE Computer Society Conference*, Boston, Massachusetts, 1971.
10. Meyer, R. A. and L. H. Seawright, "A Virtual Machine Time Sharing System," *IBM Systems Journal*, Vol. 9, No. 3, 1970.
11. Parmelee, R. P., T. I. Peterson, C. C. Tillman and D. J. Hatfield, "Virtual Storage and Virtual Machine Concepts," *IBM Systems Journal*, Vol. 11, No. 2, 1972.
12. Goldberg, R. P., *Virtual Machine Systems*, MIT Lincoln Laboratory Report No. MS-2687, (also 28L-0036), Lexington Massachusetts, 1969.
13. Mallach, E. G., "Emulation—A Survey", *Honeywell Computer Journal*, Vol. 6, No. 4, 1973.
14. Lauer, H. C. and D. Wyeth, "A Recursive Virtual Machine Architecture," *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, Cambridge, Massachusetts, 1973.
15. Scheffler, L. J., "Optimal Folding of a Paging Drum in a Three Level Memory System," *Proceedings of ACM SIGOPS Fourth Symposium on Operating Systems Principles*, Yorktown Heights, New York, 1973.