[91] Appeared in Australian Computer Science Communications 7, 1(Feb. 1985). 17.1

The Most-Recently-Used Strategy for Dynamic Storage Allocation on a Computer with Virtual Memory

Richard P. Brent
Australian National University
Box 4, Canberra, ACT 2601

Abstract

We compare several dynamic storage allocation strategies under the assumption that they will be used on a computer with virtual memory. We show that dynamic storage allocation strategies which work well on computers without virtual memory may exhibit poor paging behaviour in a virtual memory environment. We suggest a new dynamic storage allocation strategy, the "most-recently-used" (MRU) strategy, which is intended to minimize the number of page faults while keeping the total (virtual) memory used within reasonable bounds. Even in the simple case that all blocks allocated are of one fixed size, the MRU strategy may be preferable to the widely used strategy of keeping a singly-linked list of free blocks and allocating them according to a stack (i.e. last-in, first-out) discipline.

1. Introduction

Most comparisons of dynamic storage allocation strategies make the assumption that a fixed amount of random-access memory is available for use by the dynamic storage allocator [1,3,6,7,9]. If a request can not be satisfied because no sufficiently large block of memory is free, then the request either fails or is queued until it can be satisfied. Thus, the traditional criteria used to compare dynamic storage allocation strategies are:

a) the expected storage efficiency (i.e. the ratio of memory requested to memory used) under conditions of heavy loading; and

b) the efficiency of algorithms which implement the strategies (i.e. how the processor time varies with the number of blocks allocated, etc.).

In this paper we assume that a computer with "virtual" memory [5] is used. A process which executes on such a computer has a certain amount of "real" random-access memory available, but it can address a larger amount of virtual memory. When the process attempts to access a virtual memory location which does not correspond to a location in the random-access memory, a page fault interrupt occurs and a combination of hardware and software (transparent to the process) reads the page containing the virtual memory location from a secondary storage device (e.g. a disk) into random-access memory, possibly after writing out some other page to make room for it. The details of the paging process are not important here. The virtual memory available to a process is often limited by the size of the paging file rather than by the number of bits in a virtual address (see Section 2.3). If a disk access is required then the time taken to access a virtual memory location exceeds the random-access memory cycle time by a factor of more than 10,000. Hence, it is vital to ensure that most virtual memory references do not generate page faults.

Several authors have considered strategies for allocating executable code in order to minimize the number of page faults which occur when the code is executed (see for example [5,8] and the references given there). However, there does not seem to have been much study of strategies for allocating data (e.g. dynamically created records, I/O buffers, etc.) with the same objective of minimizing page faults.

On a computer with virtual memory, criterion a) above is usually of less significance than

c) the expected number of page faults which occur when the strategy is used to allocate (virtual) memory dynamically.

In this paper we compare some well-known dynamic storage allocation strategies with a new strategy, the "most-recently-used" (MRU) strategy, using criterion c). The comparison is experimental rather than theoretical because realistic theoretical results appear to be very difficult to obtain. The experimental results indicate that the MRU strategy performs better than traditional strategies on criterion c), i.e. it exhibits better paging behaviour in a virtual memory environment. A different way of stating this result is that we can get by with a smaller working set (i.e. a smaller amount of real random-access memory) for the same number of page faults if we use the MRU strategy. The MRU strategy can be implemented efficiently, i.e. it compares well with traditional strategies on criterion b). Hence, it may be of interest even if criterion c) is irrelevant.

In Section 2 we describe several strategies for dynamic storage allocation in the case that all blocks requested have the same size. This case is trivial on a computer without virtual memory, as the obvious "stack" strategy (which involves keeping a singly-linked list of free blocks and allocating them according to the last-in, first-out principle) is fast, easy to implement, and gives optimal storage efficiency. However, it is nontrivial on a computer with virtual memory: both the MRU strategy and some other strategies [4] may outperform the stack strategy on criterion c). Experimental results for the different strategies are given in Section 3.

Due to space limitations, we do not consider the case that blocks of various different sizes have to be allocated; for this case the reader is referred to [4]. However, it is easy to extend the MRU strategy to handle requests for blocks of different sizes, and the conclusion reached in [4] is essentially the same as we reach on the basis of the results quoted in Section 3: the MRU strategy (combined with another strategy such as first-fit to handle blocks larger than the page size) is preferable to well known strategies on a computer with virtual memory.

2. Allocation of equal-sized blocks

In this Section we describe four different strategies for allocating and freeing blocks of one fixed size s bytes on a computer with virtual memory and pagesize p bytes. We shall assume that the blocks are large enough to store one pointer (i.e. a virtual memory address) but significantly smaller than the page size.

2.1 The STACK strategy

A simple strategy, which we call the STACK strategy, is to keep a stack of free blocks. When a new block is requested, it is allocated from the stack according to the last-in, first-out principle. If the stack is empty, a new page of virtual memory is obtained and divided into $c = \lfloor p/s \rfloor$ blocks, which are placed on the stack. The stack may be implemented by a singly-linked list, with each free block containing a pointer to the block beneath it on the stack.

The STACK strategy is easy to implement and very efficient in term of CPU time. However, it may not be the best strategy in a virtual memory environment. To see why this might be so, consider a process which runs for a long time, allocating and freeing blocks so that each block has a

finite lifetime and the total number of blocks allocated at any time fluctuates around some equilibrium value n. (This might be the case for a discrete simulation, or for an operating system allocating and freeing I/O buffers.) After some time the addresses of blocks will be randomized so that blocks which are close to each other on the stack are unlikely to be in the same page. Thus, a large number of page faults may be generated if n/c exceeds the number of pages of real memory available to the process. Also, if n decreases after a period of high load, the number of pages referenced by the process is unlikely to decrease in proportion, because the allocated blocks will be randomly distributed over (almost) all the pages which were required in the past.

2.2 The RANDOM strategy

For the RANDOM strategy we maintain a pool of free blocks and, when a new block is requested, we randomly choose a block from the pool (unless the pool is empty, in which case we obtain a new page of virtual memory as for the STACK strategy).

We shall use the RANDOM strategy as a benchmark. A good strategy for use in a virtual memory environment should generate significantly less page faults than the RANDOM strategy! When making such comparisons we shall ignore the space overhead required to implement the RANDOM strategy efficiently (i.e. the space for an array of pointers to free blocks).

2.3 The NOFREE strategy

It is sometimes suggested that dynamic storage allocation strategies are irrelevant on computers with sufficiently large virtual address spaces because there is no need to explicitly free blocks. We call this the NOFREE strategy.

Unfortunately, the NOFREE strategy is not always feasible. The virtual pages which have not been explicitly freed must be kept either in real memory or on secondary storage (e.g. a disk), since the system has no way of telling that they will never be referenced again. Typically several processes are executing concurrently, so the quota of disk space available to each may be relatively small. For example, on the VAX computers used by the author, this quota ranges from 4 Mbyte to 14 Mbyte, much less than the 4096 Mbyte which is theoretically addressible on machines with 32-bit virtual addresses. Thus, the NOFREE strategy is not feasible for long-running processes such as operating system storage allocators, although it may be feasible for small, short-running processes. We show in Section 3 that the NOFREE strategy is not necessarily the best strategy even when it is feasible. The reason for this is that blocks which are being referenced may be spread over many pages, since most of the space in these pages is occupied by blocks which are no longer being referenced but have not been freed.

2.4 The "current page" concept

If a block has just been allocated in a page P, then it is reasonable to satisfy requests for additional blocks by allocating them in the same page P while this is possible, i.e. until all $c = \lfloor p/s \rfloor$ blocks in page P have been allocated. There are two motivations for using page P in preference to another page:

1) If a block has recently been allocated in page P then P should be in random-access memory, and allocating another block in page P should not immediately cause a page fault.

2) In many applications blocks which are allocated at about the same time tend to be referenced at about the same time. For example, a list may be created by allocating and linking several blocks; subsequently the list may be searched by following its links. Thus, it is desirable for blocks which are allocated at about the same time to be on one page or a small number of pages.

The MRU strategy and the LEAST-FREE and MOST-FREE strategies of [4] have in common that, once a "current page" P has been chosen, blocks are allocated from page P while this is possible. Once all blocks in page P have been allocated, a new current page is chosen. The strategies differ in the criteria which they use to choose the new current page.

Let $\phi(P)$ denote the number of free blocks in a page P. Since we assume that all blocks are of the same size s and are not split across page boundaries, we have $0 \leq \phi(P) \leq c$. Thus $\phi(P) = c$ means that all blocks in page P are free (although they may have been allocated and subsequently freed). We divide the set of virtual pages into four subsets:

$S_1 = \{P \mid$ no block in P has ever been allocated$\}$,

$S_2 = \{P \notin S_1 \mid \phi(P) = c\}$,

$S_3 = \{P \mid 0 < \phi(P) < c\}$, and

$S_4 = \{P \mid \phi(P) = 0\}$.

Informally, S_1 is the set of pages which have not been referenced, S_2 is the set of pages which have been referenced but contain no currently allocated blocks, S_3 is the set of pages which contain both allocated and free blocks, and S_4 is the set of pages which contain no free blocks.

A new current page can be selected from S_1, S_2 or S_3 if they are nonempty. However, the MRU strategy chooses a page in S_3 if S_3 is nonempty. If S_3 is empty but S_2 is nonempty then a page in S_2 is chosen (using the stack, i.e. last-in, first-out principle). Only if S_2 and S_3 are empty is a current page chosen from S_1. The rationale is that we want to minimize the total number of pages referenced (and hence avoid using S_1 if possible) and also minimize the number of pages on which currently allocated blocks reside (and hence avoid using S_1 or S_2 if S_3 is nonempty).

2.5 The MRU strategy

The MRU strategy uses the current page concept. Although the dynamic storage allocator does not know when a page is accessed, it does at least know when a block is allocated or freed. The MRU strategy is to maintain an ordered list of pages in S_3; when a block in page P is freed, P is moved to the head of the list (unless $\phi(P) = c$, in which case P is no longer in S_3). When it is necessary to choose a new current page, the page at the head of the ordered list is chosen (provided the list is nonempty). The rationale is that a recently referenced page is likely to be in random-access memory.

The MRU strategy can be implemented efficiently by keeping a doubly-linked list of page headers which correspond to pages in S_3: for details see [4].

3. Experimental results

In this Section we report some experimental comparisons of the four strategies for dynamic storage allocation of equal-sized blocks described

in Section 2. Since we are interested in the effect of each strategy on paging behaviour, we disregard any overheads due to the implementation of the strategies (i.e. we are comparing strategies rather than algorithms which implement them: for the distinction see [3,4]). To obtain reproducible results we simulate paging, using a strict least-recently-used paging strategy [5]. For a comparison which includes the overheads due to the implementation of the strategies and gives actual rather than simulated page faults, see [4].

3.1 Results for TRIDIAG

TRIDIAG is a Pascal program which simulates the tridiagonalization of a symmetric matrix on a systolic array [2]. In Table 3.1 we quote the page faults generated when simulating the tridiagonalization of a 128 by 128 matrix, using two different working set sizes (2000 and 4000 pages of 512 bytes). The columns headed "normalized page faults" give the ratio of page faults for each strategy to page faults for the RANDOM strategy. "virtual size" is the total number of virtual pages referenced and "working set" is the maximum number of pages in random-access memory at any one time.

TRIDIAG is typical of many list processing programs in that its usage of memory builds up to a peak and never reaches even approximate equilibrium. The block size is small (6 bytes) as each block contains only a small integer and a pointer to another block. In all cases TRIDIAG makes 1,803,213 requests to allocate a block, 1,789,653 requests to free a block, and the maximum number of allocated blocks is 545,777.

Table 3.1: Results for program TRIDIAG

Strategy	Virtual size (pages)	Page faults (working set 2000 pages)	Normalized page faults	Page faults (working set 4000 pages)	Normalized page faults
STACK	6423	3469580	0.498	606808	0.406
RANDOM	6423	6969773	1.000	1495095	1.000
NOFREE	21467	72471	0.010	21866	0.015
MRU	6423	160555	0.023	25089	0.017

Since no effort was made to make the paging simulation efficient, we do not quote CPU times in Table 3.1. However, when the paging simulation was turned off the CPU times were 1595 seconds for NOFREE, 2339 seconds for MRU, and 3093 seconds for STACK, when run on a VAX 11/750 with actual working set size 3000 pages. We were forced to abandon an attempt to compare these times with that for VAX VMS Pascal "new" and "dispose" after running for 72 hours and generating 400,000,000 page faults! For further comments on VMS routines, see [4].

From Table 3.1 we see that the STACK strategy does not perform very much better than the RANDOM strategy. NOFREE is the best strategy if the virtual size required to run it (21467 pages) is not prohibitively large. MRU is almost as good as NOFREE and requires a much smaller virtual size (6423 pages).

3.2 Results for PQSIM

PQSIM is a Pascal program which simulates discrete events using a priority queue implemented as a "leftist tree" [7]. It is probably typical of many programs which manipulate binary trees. Each dynamically allocated block (actually an unpacked Pascal record) contains five fields: two pointers,

a key, a priority value, and a small integer giving the distance to the nearest leaf in the tree. Thus, the block size is 20 bytes.

In Table 3.2 we compare the paging behaviour of several dynamic storage allocation strategies when used by PQSIM. The events simulated by PQSIM had exponentially distributed inter-arrival times and lifetimes. The mean number of arrivals per (simulated) second was about 200 and at equilibrium the mean number of events in the queue was about 2400: for details see [4]. In all cases the simulated working set size (excluding implementation-dependent overheads) was 50 pages. PQSIM was run for 2000 (simulated) seconds, but page faults were only counted after the first 1000 (simulated) seconds, once approximate equilibrium had been reached. Results quoted for NOFREE in Tables 3.2 and 3.3 are estimated, due to paging file quota restrictions.

Table 3.2: Results for program PQSIM

Strategy	Virtual size (pages)	Page faults	Normalized page faults
STACK	101	561080	0.71
RANDOM	102	794000	1.00
NOFREE	31360	445709	0.56
MRU	101	374908	0.47

From Table 3.2 we see that the difference between the strategies for PQSIM is not so marked as for TRIDIAG. Even so, the best strategy (MRU) generates less than 50% of the number of page faults that the RANDOM strategy does. MRU is significantly better than NOFREE and STACK. Thus, the NOFREE strategy is not always best even if it is feasible. (For more evidence of this, see [4].)

3.3 The effect of varying the load

In some applications it is desirable to have a strategy which adapts well to a slowly changing load. For example, we expect the response of a time-sharing system to improve when the load decreases. Thus, the simulations described in Section 3.2 were repeated with the load for the first 1000 simulated seconds doubled (i.e. mean inter-arrival times halved). The load for the second period of 1000 simulated seconds, during which page faults were counted, was the same as before. The results are summarized in Table 3.3. The last column of the table gives the ratio (page faults with initial load doubled)/(page faults with initial load as in Section 3.2).

Table 3.3: Results for PQSIM after initial load doubled

Strategy	Page faults	Normalized page faults	Page fault ratio
STACK	1044235	0.76	1.86
RANDOM	1372438	1.00	1.73
NOFREE	520717	0.38	1.17
MRU	337648	0.25	0.90

From Table 3.3 we see that the STACK and RANDOM strategies do not adapt well to a decreasing load. This is because blocks are spread over a large number of pages when the load is high, and most of these pages continue being accessed after the load has declined. The NOFREE strategy adapts reasonably well, in that it performs almost as well after a period of abnormally high load as after a period of normal load. The MRU strategy

adapts very well, in fact it performs better after a period of high load than after a period of normal load. Overall the MRU strategy performs significantly better than the other strategies.

3.4 Comments on the experimental results

From the experimental results quoted in Sections 3.1-3.3, and similar results given in [4] but not quoted here because of space limitations, we conclude:

1. The NOFREE strategy performs well, but not always as well as the MRU strategy (see Tables 3.2 and 3.3). Thus, even when virtual size restrictions do not preclude the use of the NOFREE strategy, it is not necessarily the best choice.

2. The MRU strategy performs better than the STACK strategy, and the difference can be dramatic (see Table 3.1). Hence, the STACK strategy is not to be recommended for use in a virtual memory environment.

3. Other strategies which use use "current page" concept (Section 2.4) perform well, but the MRU strategy appears to be the best of the strategies considered in [4]. This may depend on the paging strategy - it is plausible that our choice of the least-recently-used paging strategy biases our results in favour of MRU.

4. Conclusion

The criteria which should be used to compare dynamic storage allocation strategies (and the algorithms which implement the strategies) differ if the strategies are to be used on a computer with virtual memory rather than on one without virtual memory. We have presented a new strategy, the MRU strategy, which is intended to be good on a computer with virtual memory. The new strategy can be implemented efficiently and has good worst-case behaviour (see [4]). It compares well with traditional strategies, both in the special case that all blocks have the same size (see Section 3) and in the general case where blocks may differ in size (see [4]). Even on a computer without virtual memory, the MRU strategy may be a good choice because it can be implemented so that blocks can be allocated and freed in constant time (independent of the total number of allocated blocks) and gives high storage efficiency (see [4]). Because the MRU strategy is only applicable for blocks of size less than the page size, it has to be combined with another strategy which is used for large blocks. The first-fit strategy is a suitable choice, provided that it is implemented well [3,4].

None of the strategies considered in this paper make use of information about which virtual pages are present in random-access memory, and better strategies might be feasible if such information is available. Another area for future work is the influence of different paging strategies.

The MRU strategy has been implemented in (slightly machine-dependent) Pascal on a VAX computer running under VMS; readers interested in obtaining further information should contact the author.

5. Acknowledgement

The support of the Australian Research Grants Scheme and the Centre for Mathematical Analysis at the Australian National University is gratefully acknowledged.

6. References

1. C. Bays, A comparison of next-fit, first-fit and best-fit, *Comm. ACM* 20 (1977), 191-192.

2. A. Bojanczyk and R. P. Brent, Tridiagonalization of a symmetric matrix on a square array of mesh-connected processors, to appear in *J. Parallel and Distributed Computing*. Available as Report CMA-R45-83, Centre for Mathematical Analysis, Australian National University, December 1983.

3. R. P. Brent, Efficient implementation of the first-fit strategy for dynamic storage allocation, *Australian Computer Science Communications* 3 (1981), 25-34. Revision available as Report CMA-R33-84, Centre for Mathematical Analysis, Australian National University, August 1984.

4. R. P. Brent, Dynamic storage allocation on a computer with virtual memory, Report CMA-R37-84, Centre for Mathematical Analysis, Australian National University, September 1984.

5. P. J. Denning, Virtual memory, *Computing Surveys* 2 (1970), 153-189.

6. J. B. Hext, A storage management laboratory, *Australian Computer Science Communications* 2 (1980), 185-193.

7. D. E. Knuth, *The Art of Computer Programming*, Vol. 1 (2nd edition), Addison-Wesley, Reading, Mass., 1973.

8. J. W. Stamos, Static grouping of small objects to enhance performance of a paged virtual memory, *ACM Transactions on Computer Systems* 2 (1984), 155-180.

9. C. J. Stephenson, Fast fits - New methods for dynamic storage allocation, *Proc. Ninth ACM Symposium on Operating System Principles*, ACM, New York, 1983, 30-32. Summary of a paper to appear in *ACM Transactions on Computer Systems*.