DYNAMIC STORAGE ALLOCATION ON A COMPUTER WITH VIRTUAL MEMORY

R.P. Brent

CMA-R37-84

Centre for Mathematical Analysis
Hanna Neumann Building
Australian National University
GPO Box 4
Canberra ACT 2601
Australia

ABSTRACT

We compare several dynamic storage strategies under the assumption that they will be used on a computer with virtual memory. On such a computer the total (virtual) memory referenced by a process may exceed the actual amount of random-access memory available to the process, and it is usually more important to minimize the number of page faults than the virtual memory required by the process. 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 some new dynamic storage allocation strategies which are intended to minimize the number of page faults while keeping the total (virtual) memory used within reasonable bounds. The new strategies can be implemented efficiently and are preferable to well-known strategies such as the first-fit strategy if the blocks being allocated are appreciably smaller than the page size. Even in the simple case that all blocks are of one fixed size, the new strategies 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.

CR Categories and Subject Descriptors:

D.3.4 Programming Languages: Processors - run-time environments;

D.4.2 Operating Systems: Storage Management - allocation/deallocation s strategies, main memory, virtual memory;

D.4.8 Operating System: Performance - modelling and prediction, simulation;

E.2 Data: Data Storage Representations - contiguous representations.

General Terms: Algorithms, Performance.

Additional Key Words and Phrases: Dynamic storage allocation, dynamic memory management, paging, virtual memory, list processing, linked lists, stacks, first-fit strategy, least-free strategy, most-free strategy, most-recently-used strategy, new, dispose, Pascal.

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 [1,5,7,10,18]. If a request cannot 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 [6] 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. As a concrete example, the VAX 11/750 computer used to obtain the results of Sections 3 and 7 has a page size of 512 bytes, the maximum amount of real random-access memory available to a process (i.e. the maximum working set size) is typically in the range 200 to 3000 pages, the actual random-access memory size is 6144 pages, and the maximum amount of virtual memory available to a process is 28672 pages, i.e. 14 Mbytes. The virtual memory size is limited by the size of the paging file rather than by the number of bits in a virtual address (see Section 2). 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 10^4 . Hence, it is vital to ensure that most virtual memory references do not generate page faults. (Sometimes a page fault may not necessitate a disc access because the operating system may keep a list of recently-used pages in random-access memory, but we shall ignore this complication below.)

Several authors have considered strategies for allocating executable code in order to minimize the number of page faults which occur when a process is executed (see for example [6,17] 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 several well-known dynamic storage allocation strategies and some new strategies, using criterion c) and, to a lesser extent, criteria a) and b). The comparison is mainly experimental rather than theoretical, because realistic theoretical results appear to be very difficult to obtain. The experimental results indicate that the new strategies perform better than traditional strategies on criterion c), i.e. they exhibit better paging behaviour in a virtual memory environment. A different way of stating this result is that if one of the new strategies is used instead of one of the traditional strategies then 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. The new strategies can also be implemented efficiently, i.e. they compare well with the traditional strategies on criterion b). Hence, they may be of interest even if criterion c) is irrelevant.

In Section 2 we consider 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, even this simple case is nontrivial on a computer with virtual memory: both our new strategies and the first-fit strategy may outperform the stack strategy on criterion c). Some examples are given in Section 3.

In Section 4 we outline how our new strategies can be implemented by efficient algorithms. Our distinction between a dynamic storage allocation *strategy* and an *algorithm* which implements the strategy is the same as in [5]. In Section 5 we show how the new strategies and the algorithms which implement them can easily be extended to handle the case of variable block size, subject to the constraint that the maximum block size should not exceed the page size. Theoretical worst-case bounds on storage efficiency are discussed in Section 6, and experimental results for the variable block size algorithms are given in Section 7.

Our conclusions are stated in Section 8. To summarize, we recommend a combination of one of our new strategies (for blocks significantly smaller than the page size) and the first-fit strategy (for larger blocks).

The ideas behind our new strategies may be used in other applications where locality of storage references is desirable. For example, when allocating files on a disk we may wish to minimize the number of disk head movements required to access the files. All files stored in one disk cylinder can be accessed with at most one head movement, just as all memory locations in one page can be accessed with at most one page fault. The time required to move the disk head to another cylinder may be several times longer than the rotation time of the disk (but the ratio is much less than the value 10^4 mentioned above). Hence, our new strategies might be useful for allocating space on a disk if "byte" (or "word") is replaced by "sector" (or "track") and "page" is replaced by "cylinder".

2. ALLOCATION OF EQUAL-SIZED BLOCKS

In this Section we consider a simple but common special case: the allocation of blocks of one fixed size s bytes on a machine with virtual memory and page size 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. First we describe ten different strategies for allocating and freeing blocks of size s, then (in Section 3) we present the results of some experimental comparisons of the different strategies.

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 terms 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 QUEUE Strategy

The "queue" strategy is very similar to the stack strategy, the difference being that blocks are allocated according to the first-in, first-out principle (instead of last-in, first-out). The queue strategy may be implemented efficiently with a singly linked list so long as pointers to both ends of the list are maintained. In a virtual memory environment the queue strategy suffers the same disadvantages as the stack strategy. In fact, we might expect the queue strategy to generate more page faults than the stack strategy because a block which has recently been freed is more likely to be in a page which is in real memory than is a block which was freed less recently.

2.3 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 and queue strategies). The random strategy can be implemented with reasonable efficiency by maintaining an array of pointers to free blocks.

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 overheads required to implement the random strategy efficiently (i.e. the array of pointers to free blocks) since approximations to random behaviour can be obtained with much lower space overhead. Note that our random strategy is not identical to the "random fit" strategy of Reeves [12].

2.4 The NOFREE Strategy

It it sometimes suggested that dynamic storage allocation strategies are irrelevant on computers with a sufficiently large virtual address space 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.

2.5 The QUICK Strategy

The "quick" strategy is an attempt to retain the simplicity of the nofree strategy while avoiding its major disadvantage of using an unbounded number of pages even if the total number of blocks allocated but not freed remains bounded. With each page we maintain a count of the number of blocks allocated but not freed in the page. When this count drops to zero we add the page to a stack of free pages which are used in preference to new virtual pages.

The quick strategy is easy to implement and almost as fast as the nofree strategy. Although it gives low storage efficiency, it is often feasible when the nofree strategy is not. For a comparison of its paging behaviour with that of other strategies, see Sections 3 and 7.

2.6 The FIRST-FIT Strategy

The "first-fit" strategy [10] is well known as a strategy for dynamic storage allocation of unequal-sized blocks but it can, of course, also be used for equal-sized blocks. The first-fit strategy is simply to allocate each block at the lowest possible virtual address. Note that this may result in blocks being split across page boundaries, which is not the case for the other strategies described in this Section.

It appears to be impossible to implement the first-fit strategy as efficiently as the other strategies described in this Section. A straightforward implementation [4,10] takes time $O(F)$ on average to allocate and free a block, where F is the total number of free blocks; better implementations [5] take time $O(\log F)$. The other strategies described in this Section can be implemented so the average time required to allocate and free a block is constant, independent of the number of allocated/free blocks (for details see Section 4).

Implementations of the first-fit strategy may have higher space overheads than implementations of competing strategies. For example, the first-fit implementation of [5] requires one additional pointer per block. This overhead is significant if the block size is small. In Section 3 we are interested in comparing the paging behaviour induced by different strategies, so we shall not consider the space and time overheads required to implement the strategies, but the reader should bear in mind that this gives the first-fit strategy an unfair advantage. Even with this advantage, the first-fit strategy does not turn out to be the best strategy.

2.7 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 strategies described in Sections 2.8 - 2.11 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. As far as we know these strategies are new.

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 $0 \leq \phi(P) \leq c = \lfloor p/s \rfloor$. Thus $\phi(P) = 0$ means that there are no free blocks in page P , while $\phi(P) = c$ 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:

$$S_1 = \{P \mid \text{no block in } P \text{ has ever been allocated}\} ,$$
$$S_2 = \{P \notin S_1 \mid \phi(P) = c\} ,$$
$$S_3 = \{P \mid 0 < \phi(P) < c\} , \text{ and}$$
$$S_4 = \{P \mid \phi(P) = 0\} .$$

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 four strategies described in the remainder of this Section all choose a page in S_3 if S_3 is nonempty. If S_3 is empty but S_2 is nonempty then they choose a page in S_2 (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). The four strategies differ in how they choose a new current page from S_3 when S_3 is nonempty.

2.8 The RANDOM(B) Strategy

The "random(B)" strategy provides a benchmark: if S_3 is nonempty we randomly select a page in S_3 (with equal probabilities $1/|S_3|$) to be the new current page. Once the current page has been selected we allocate blocks from it as described in Section 2.7 (compare the "random" strategy of Section 2.3).

2.9 The MOST-FREE Strategy

The "most-free" strategy chooses a page $P \in S_3$ with maximal $\phi(P)$ as the new current page. In case of a tie, the stack principle is used. To implement the most-free strategy efficiently we need to maintain a stack of pages for each possible value of $\phi(P)$ in the range $0 < \phi(P) < c$: for details see Section 4.

The motivation for the most-free strategy is that it locally minimizes the number of different pages which are used to satisfy a sequence of requests for blocks which might be referenced together (see Section 2.7).

2.10 The LEAST-FREE Strategy

The "least-free" strategy chooses a page $P \in S_3$ with minimal $\phi(P)$ as the new current page; otherwise it is identical to the most-free strategy. There are two motivations for the least-free strategy:

1) In the absence of other information it is reasonable to assume that a page P with a larger number of allocated blocks (i.e. a smaller value of $\phi(P)$) is more likely to be accessible without generating a page fault than is a page with a smaller number of allocated blocks.

2) By choosing a page P with small $\phi(P)$ as the current page we make it more likely that pages $Q \in S_3$ with larger values of $\phi(Q)$ will migrate to S_2 as the allocated blocks in them are freed. Thus, the least-free strategy should adapt well to a slowly fluctuating load (where the "load" is the number of blocks which have been allocated but not freed).

2.11 The MRU Strategy

The final strategy which uses the concept of a current page is the "most recently used" strategy (abbreviated "MRU"). The dynamic storage allocator does not know when a page is accessed, but 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 a page P is freed, P is moved to the head of the list (unless $\phi(P) = c$, in which case $P \in S_2$). When it is necessary to choose a new current page, the page at the head of the ordered list is chosen. The rationale for this strategy is that a recently referenced page is likely to be in random-access memory.

2.12 Using Additional Information

None of the strategies described above require the dynamic storage allocator to know which pages currently reside in random-addess memory, although some of the strategies are motivated by the likelihood of certain pages (e.g. recently used pages) being in random-access memory. Obvious modifications of the strategies are possible if it can be determined (without generating a page fault) whether a given page P is present in random-access memory. However, we shall not consider such modifications in this paper.

2.13 The Tolerance ϕ_0

The random(B), least-free, most-free and MRU strategies may be generalized by the introduction of a positive tolerance $\phi_0 < c$. When choosing a new current page from S_3, we consider only those pages P for which $\phi(P) \ge \phi_0$. The motivation for the introduction of the tolerance ϕ_0 is similar to the motivation for the most-free strategy: once a current page is chosen we want to allocate several blocks from it. The optimal choice of ϕ_0 is considered in Section 3.3. If ϕ_0 is not specified it is assumed to be 1 (as in Sections 2.7 to 2.11 above).

3. EXPERIMENTAL RESULTS FOR EQUAL-SIZED BLOCKS

In this Section we report some experimental comparisons of the ten 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 (see Section 2.6). To obtain reproducible results we simulate paging, using a strict least-recently-used paging strategy [6].

For a comparison which includes the overheads due to the implementation of the strategies and gives actual rather than simulated page faults, see Section 7.

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 512-byte pages). 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 storage builds up to a peak and never reaches even approximate equilibrium. The block size is small (6 bytes) as blocks contain 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.

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 ranged from 1595 sec (for the NOFREE strategy) to 5693 sec (for the FIRST-FIT strategy) and were about 2500 sec for the MOSTFREE, LEASTFREE and MRU strategies, when run on a VAX 11/750 with actual working set size of 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 Section 7.3.)

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
QUEUE	6423	3048212	0.437	460270	0.308
RANDOM	6423	6969773	1.000	1495095	1.000
NOFREE	21467	72471	0.010	21866	0.015
QUICK	9361	163520	0.024	29721	0.020
FIRST-FIT	6397	171484	0.025	30593	0.020
RANDOM(B)	6423	165488	0.024	27374	0.018
MOST-FREE	6423	172207	0.025	25589	0.017
LEAST-FREE	6423	172765	0.025	24768	0.017
MRU	6423	160555	0.023	25089	0.017

From Table 3.1 we see that the STACK and QUEUE strategies do not perform very much better than the RANDOM strategy. NOFREE is the best strategy if it is feasible (i.e. if it does not give a prohibitively large virtual size). The other six strategies (QUICK, FIRST-FIT, RANDOM(B), MOST-FREE, LEAST-FREE and MRU) give approximately the same number of page faults, slightly more than NOFREE but much less than RANDOM, STACK or QUEUE. The QUICK strategy gives a virtual size about 50 percent larger than FIRST-FIT etc. but much smaller than NOFREE.

3.2 Results for PQSIM

PQSIM is a Pascal program which simulates discrete events using a priority queue implemented as a "leftist tree" [10]. 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 various 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 more details, see the description of the CAMBRIDG distribution in Section 7.1.) 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 (working set 50 pages)	Normalized page faults
STACK	101	561080	0.71
QUEUE	101	803555	1.01
RANDOM	102	794000	1.00
NOFREE	31360	445709	0.56
QUICK	1823	531122	0.67
FIRST-FIT	98	455026	0.57
RANDOM(B)	103	606035	0.76
MOST-FREE	102	453102	0.57
MOST-FREE(ϕ_0=5)	118	361515	0.46
LEAST-FREE	102	511869	0.64
LEAST-FREE(ϕ_0=5)	116	377282	0.48
MRU	101	374908	0.47
MRU(ϕ_0=5)	118	337436	0.42

From Table 3.2 we see that the difference btween the strategies is not so marked as for the program TRIDIAG (see Table 3.1). Even so, the better strategies generate less than half as many page faults as the RANDOM strategy does. MRU, MOST-FREE (with $\phi_0 = 5$) and LEAST-FREE (with $\phi_0 = 5$) are significantly better than NOFREE. Thus, the NOFREE strategy is not always best even if it is feasible.

3.3 The Effect of Varying 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, we repeated the simulations described in Section 3.2 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 (initial load 2)	Normalized page faults	Page fault ratio
STACK	1044235	0.76	1.86
QUEUE	1365741	1.00	1.70
RANDOM	1372438	1.00	1.73
NOFREE	520717	0.38	1.17
QUICK	597452	0.44	1.12
FIRST-FIT	643004	0.47	1.41
RANDOM (B)	568352	0.41	0.94
MOST-FREE	498482	0.36	1.10
MOST-FREE (ϕ_0=5)	477215	0.35	1.32
LEAST-FREE	726429	0.53	1.42
LEAST-FREE (ϕ_0=5)	523993	0.38	1.39
MRU	337648	0.25	0.90
MRU (ϕ_0=5)	361058	0.26	1.07

From Table 3.3 we see that the STACK, QUEUE 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, QUICK, FIRST-FIT, MOST-FREE and LEAST-FREE strategies adapt reasonably well, in that they perform almost as well after a period of abnormally high load as they do after a period of normal load (the ratios given in the last column of the table are in the range 1.08 to 1.42). The RANDOM(B) and MRU ($\phi_0 = 1$) strategies adapt very well: the ratios given in the last column of the table are less than unity, i.e. the strategies perform 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 The Choice of ϕ_0

In Tables 3.2 and 3.3 we gave results for the MOST-FREE, LEAST-FREE and MRU strategies for two different values of the tolerance ϕ_0 (see Section 2.13). The strategies performed better with $\phi_0 = 5$ than with the default value of $\phi_0 = 1$. (This is not always true: for the program TRIDIAG of Section 3.1 there was no significant difference between the results with $\phi_0 = 1$ and with $\phi_0 > 1$.) In Table 3.4 we show the effect of different choices of ϕ_0 for the MOST-FREE (MF), LEAST-FREE (LF) and MRU strategies. Apart from the choice of ϕ_0 everything is as in Section 3.2. Note that $c = \lfloor p/s \rfloor = 25$ so $1 \leq \phi_0 \leq 25$.

Table 3.4: Results for PQSIM with various ϕ_0

ϕ_0/c	virtual size MF	virtual size LF	virtual size MRU	normalized page faults MF	normalized page faults LF	normalized page faults MRU
0.04	102	102	101	0.57	0.64	0.47
0.08	104	105	107	0.52	0.57	0.51
0.12	108	110	110	0.51	0.54	0.48
0.16	113	111	111	0.48	0.48	0.42
0.20	118	116	118	0.46	0.48	0.42
0.24	123	124	124	0.48	0.47	0.44
0.28	131	130	128	0.45	0.47	0.44
0.32	132	137	131	0.44	0.46	0.43
0.36	141	144	140	0.44	0.45	0.44
0.40	151	152	149	0.44	0.46	0.43
0.44	161	164	161	0.44	0.47	0.43
0.48	171	171	171	0.45	0.45	0.44
0.52	183	179	181	0.46	0.45	0.45
0.56	189	201	194	0.45	0.46	0.47
0.60	214	212	211	0.45	0.45	0.45
0.64	228	232	222	0.47	0.46	0.45
0.68	256	244	251	0.47	0.48	0.46
0.72	282	282	279	0.48	0.47	0.47
0.76	307	320	312	0.47	0.49	0.48
0.80	349	357	351	0.48	0.50	0.49
0.84	423	407	412	0.50	0.50	0.50
0.88	496	494	501	0.52	0.51	0.50
0.92	651	631	642	0.52	0.52	0.52
0.96	965	932	925	0.54	0.54	0.53
1.00	1635	1622	1740	0.54	0.54	0.54

From Table 3.4 we see that the behaviour of the MOST-FREE, LEAST-FREE and MRU strategies is very similar, with MRU performing marginally better than the others. The number of page faults as a function of ϕ_0 has a minimum at some optimal value $\phi_{opt} \simeq 0.4c$. However, the dependence on ϕ_0 is very small (i.e. the graph is very flat) for $0.16 \leq \phi_0/c \leq 0.6$. As expected, the virtual size increases with ϕ_0 . Thus, we may prefer to choose $\phi_0 < \phi_{opt}$ in order to reduce the virtual size at the expense of a slight increase in the number of page faults. See also Section 7.3.

3.5 Conclusions

From the experimental results quoted in Sections 3.1 to 3.4, and similar results which are not quoted because of space limitations, we may draw some conclusions:

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

2. The QUICK strategy performs slightly worse than the NOFREE strategy, but is more likely to be feasible because of its significantly smaller virtual size requirements.

3. Strategies which use the "current page" concept (see Section 2.7) generally perform better than the STACK and QUEUE strategies, and the difference can be dramatic (see Table 3.1). Hence, we do not recommend the STACK or QUEUE strategies for use in a virtual memory environment.

4. The MRU, MOST-FREE and LEAST-FREE strategies (with a good choice of ϕ_0) can perform significantly better than the RANDOM(B) strategy, so their performance is not entirely due to their use of the "current page" concept. They perform significantly better than the FIRST-FIT strategy (and, as shown in Section 4, can also be implemented more efficiently than FIRST-FIT). Overall the MRU strategy appears to be the best choice, although the differences between MRU, MOST-FREE and LEAST-FREE are small and may depend on the paging strategy. It is plausible that our choice of the least-recently-used paging strategy favours the MRU storage allocation strategy.

4. EFFICIENT IMPLEMENTATION OF THE STRATEGIES

In this Section we describe how the MOST-FREE, LEAST-FREE and MRU strategies can be implemented so that requests for the allocation and release of blocks can be met in constant time (independent of the total number of allocated or free blocks). We still assume that all blocks are of the same size $s \le p$, and that virtual memory can be allocated in units of one page. First consider the MOST-FREE and LEAST-FREE strategies.

Within each page we allocate $c = \lfloor p/s \rfloor$ blocks using the STACK strategy (or any other reasonable strategy - clearly the strategy for allocating blocks within each page does not influence the paging behaviour). For simplicity we do not attempt to split blocks across page boundaries. Each page P in use has an associated "page header" record $H(P)$ containing the following fields:

next, prev:	pointers to adjacent page headers in a doubly-linked list;
leftmost:	pointer to "left header" for doubly linked list (see below);
freelist:	pointer to top of stack of free blocks within the page;
freekt:	the number of free blocks within the page.

The page headers for pages having the same number ϕ of free blocks are kept in a doubly linked list L_ϕ (hence the "next" and "prev" fields). For ease of implementation each such list is circular. For each list L_ϕ there is a "left header" record LH_ϕ containing the following fields:

next, prev:	pointers to page headers in L_ϕ ;
up, down:	pointers to left headers;
s:	the block size (only necessary when several block sizes are permitted: see Section 5).

The left headers LH_ϕ are kept in a doubly-linked list ordered by their ϕ values (hence the "up" and "down" fields). The whole data structure is accessible via a pointer to LH_0 (which is always present, even if L_0 is empty).

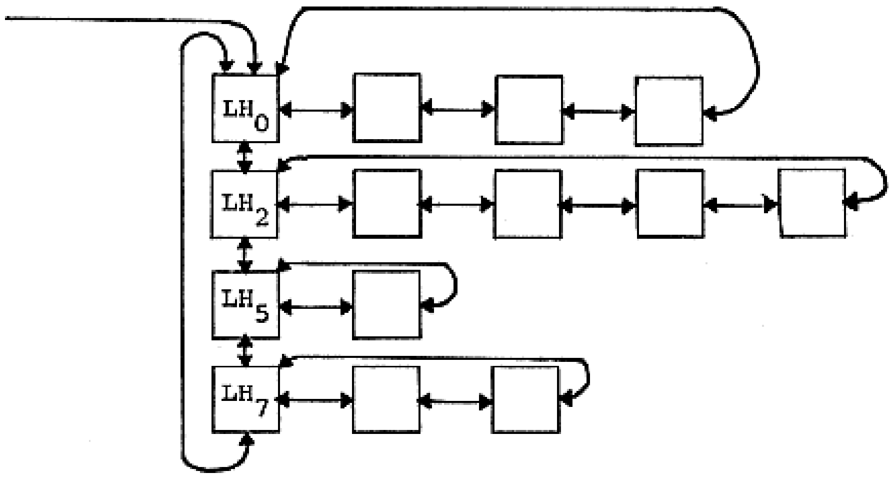

Figure 4.1: Illustration of the data structure for page headers.

Apart from LH_0, the left headers LH_ϕ are omitted if their corresponding lists L_ϕ are empty. This is illustrated in Figure 4.1, where four lists (L_0, L_2, L_5 and L_7) of page headers are nonempty.

When a block in a page P is freed, the corresponding page header H(P) has to be removed from its list L_ϕ and inserted in $L_{\phi+1}$. This is straightforward and can be done in constant time, independent of the length of L_ϕ, because LH_ϕ can be accessed via the "leftmost" field in H(P). (The leftmost fields are not shown in Figure 4.1.)

When a block is freed only its virtual address need be known. From its address it is easy to compute the index of the page P containing it, and then to locate the corresponding page header H(P), provided that each page P which is in use contains a pointer to its header record H(P). It is not desirable to store the header record H(P) in page P, because updating the doubly-linked lists of page headers might then generate more page faults than if the headers and left headers were stored in dedicated pages which would usually be present in random-access memory.

In order to allocate a block, the "current page" C must be accessible. An efficient solution is to keep the header $H(C)$ for the current page at the head of the list L_0, so it is accessible via the left header LH_0. (When a block in C is freed, the header $H(C)$ is not moved, but the "freekt" field of $H(C)$ is incremented.) To allocate a block, we check if the "freekt" field of $H(C)$ is positive. If so, the block can be allocated on page C and the "freekt" field decremented. Otherwise a new current page has to be selected. For the MOST-FREE strategy we follow the "up" field of the left header LH_0 to find the left header LH_ϕ with maximal $\phi < c$. For the LEAST-FREE strategy we follow the "down" field to find the left header LH_ϕ with minimal $\phi \geq \phi_0$. In order to avoid scanning a list of length $O(\phi_0)$, it is desirable to merge the lists $LH_0, LH_1, \ldots, LH_{\phi_0-1}$, i.e. the page headers for pages with less than ϕ_0 free blocks are all kept on the same doubly-linked list LH_0. This also decreases the number of doubly-linked list insertions and deletions required when blocks are freed.

For the MRU strategy we merge the doubly-linked lists $LH_{\phi_0}, \ldots, LH_{c-1}$, i.e. the page headers for pages with ϕ free blocks, where $\phi_0 \leq \phi < c$, are all kept on the same doubly-linked list LH_{ϕ_0}. When a block in such a page is freed, the corresponding header is moved to the head of LH_{ϕ_0}. Otherwise the MRU strategy is implemented in the same way as the LEAST-FREE strategy.

Other efficient implementations of the three strategies are possible. For example, the left headers might be maintained in an array with indices $0..c$ instead of a doubly-linked list, and the page header $H(P)$ corresponding to a block in page P might be found by hashing the index of P instead of by reserving a pointer in each page (the results quoted in Section 7.2 are for an implementation which uses hashing in this way). The important point is that the strategies can be implemented so that requests for the allocation and release of blocks can be met in constant time. Thus the strategies are competitive with popular strategies such as first-fit, best-fit, etc. if CPU time is the dominant criterion.

5. EXTENSION TO BLOCKS OF DIFFERENT SIZES

So far we have restricted our attention to the case that all blocks requested have the same size s. However, it is easy to extend the MOST-FREE, LEAST-FREE and MRU strategies to handle requests for blocks of arbitrary size $s \leq p$. We need a mapping SIZEMAP (which will be implemented as an array) from the set of possible sizes $1..s_{max}$ to itself, satisfying the following condition:

a) $\text{SIZEMAP}[s] \geq s$ for $1 \leq s \leq s_{max}$.

In addition, a practical mapping would be monotonic, i.e.

b) $\text{SIZEMAP}[s+1] \geq \text{SIZEMAP}[s]$ for $1 \leq s < s_{max}$.

Because of memory alignment constraints, we might also insist that SIZEMAP[s] be divisible by some small power of 2 (e.g. 4 or 8). Alignment constraints will be ignored, since they are easily handled by changing the units in which storage is measured (e.g. from bytes to words). We discuss the choice of the mapping SIZEMAP in more detail in Sections 5.1 and 5.3.

A request for a block of size s is satisfied by allocating a block of (possibly larger) size SIZEMAP[s]. For each distinct value in the range of SIZEMAP, we use the MOST-FREE, LEAST-FREE or MRU strategies, as described in Sections 2 and 4. The only significant change is that instead of having a single pointer to the left header LH_0 (see Section 4) we now have an array of pointers, because there is now a distinct left header LH_0 for each distinct block size SIZEMAP[s]. (A less significant change is that the headers for pages in which all blocks are free are kept on a single linked list, instead of one list for each possible block size. Thus, at any given time a page contains blocks of only one size, but that size may change if all the blocks allocated in the page are freed.)

5.1 The choice of SIZEMAP

Suppose that there are k distinct values $s_1 < s_2 < \ldots < s_k$ in the range of SIZEMAP, so SIZEMAP[s]= s_1 if $1 \leq s \leq s_1$, and SIZEMAP[s] = s_j if $s_{j-1} < s \leq s_j$, $1 < j \leq k$. We shall consider how $s_1, \ldots, s_k$ should be chosen to give a reasonably high storage efficiency. The worst case is considered in Section 6.

Consider two possible block sizes s' and s''. If $\lfloor p/s' \rfloor = \lfloor p/s'' \rfloor$ then there is no advantage to be gained in making SIZEMAP[s'] and SIZEMAP[s"] distinct. Thus, a simple algorithm for choosing $s_1, s_2, \ldots, s_k$ is:

```
$s_1$ := minimum useful block size;
$j$ := 1 ;
while $s_j < p$ do
   begin
   $s_{j+1}$ := max $(s_j + 1, \lfloor p/(\lfloor p/s_j \rfloor - 1) \rfloor)$ ;
   $j$ := $j + 1$
   end;
$k$ := $j$ .
```

If $s_1, \ldots, s_k$ are chosen in this way, it is easy to see that

$$s_{j+1} = s_j + 1 \quad \text{for} \quad s_j \leq p^{\frac{1}{2}}$$

and

$$\lfloor p/s_{j+1} \rfloor = \lfloor p/s_j \rfloor - 1 \quad \text{for} \quad s_j \geq p^{\frac{1}{2}} ,$$

so

$$k \simeq 2p^{\frac{1}{2}} .$$

5.2 The parameter FREETOL

When considering the allocation of blocks of one fixed size, we assumed that the page capacity c and tolerance ϕ_0 (Section 2.13) were fixed. Now, with blocks of several sizes, we have different capacities $c_j = \lfloor p/s_j \rfloor$ for each size s_j, $j = 1..k$. It seems reasonable to take the tolerance ϕ_0 proportional to c_j rather than constant. Thus, define FREETOL to be the number of bytes which must be free in a page before it is considered as a candidate for the next current page (Sec. 2.13), and for each block size s_j take $\phi_0(s_j) = \lceil \text{FREETOL}/s_j \rceil$. The results of Section 3.4 suggest that $0.4p$ is a reasonable choice for FREETOL (but see also the comments at the end of Section 7.3).

5.3 Storage efficiency and the parameter FFCROSSOVER

As in Section 1, we define "storage efficiency" to be the ratio

$$E = \frac{\text{total size of blocks requested but not freed}}{\text{total storage used to satisfy requests}} .$$

Even if blocks are never freed after being allocated, the storage efficiency E will be less than 1, for the following reasons:

a) "Internal fragmentation" [15,16] due to rounding up a request of size s to $s_j = \text{SIZEMAP}[s] \geq s$;

b) "External fragmentation" [15,16] due to the loss of $(p \bmod s_j)$ bytes in each page in which blocks of size s_j are allocated, since blocks are not split across page boundaries;

c) "External fragmentation" due to allocation of an integral number of pages for each size s_j ; and

d) Space occupied by page headers and left headers, etc.

The loss due to a) and b) combined when several requests of size s are satisfied is less than s bytes per page if SIZEMAP is chosen as above. Thus, a) and b) do not seriously reduce the storage efficiency so long as $s \ll p$. The worst case is $s = \lfloor p/2 \rfloor + 1$, when only about 50 percent of each page is used.

The loss due to c) is at most one page per distinct block size allocated, i.e. at most $k \simeq 2p^{\frac{1}{2}}$ pages in all. This is not serious if a large number of pages are allocated, but may be serious if the number of pages allocated is small. (Compare the storage efficiencies for the CAMBRIDG and YKTVMV simulations of Section 7.2.)

Finally, the loss due to d) is relatively small so long as the page header and left header records are much smaller than one page. For our implementation on a VAX the page headers occupy 20 bytes (24 if hashing is used) which is less than 5 percent of the page size.

We conclude that the MOST-FREE, LEAST-FREE and MRU strategies may give high storage efficiencies so long as the maximum block size requested is significantly smaller than the page size. On the other hand, the FIRST-FIT strategy implemented as in [5] gives high storage efficiency so long as the block sizes are not too small (because there is a certain fixed overhead per block). Thus, it is reasonable to combine the MOST-FREE, LEAST-FREE or MRU strategies for blocks of size $s \leq$ FFCROSSOVER with the FIRST-FIT strategy for blocks of size $s >$ FFCROSSOVER, where FFCROSSOVER $< p$ is a suitably chosen threshold. A reasonable choice is FFCROSSOVER $\simeq p/4$ (see Section 7).

If we know the distribution of block size requests (as is the case in Section 7, for example) and FFCROSSOVER has been chosen, then a dynamic programming approach can be used to choose a close to optimal SIZEMAP in order to (almost) maximise the expected storage efficiency. The basic idea is, for each possible size $s = \text{FFCROSSOVER}-1$ down to s_1, we decide whether to choose SIZEMAP[s] = s or SIZEMAP[s] = SIZEMAP[s+1] after estimating the expected storage required from a) to d) above. We shall omit the details.

6. WORST-CASE FRAGMENTATION FOR THE MOST-FREE STRATEGY

In this Section we consider the worst-case storage fragmentation for the MOST-FREE strategy when applied to blocks of arbitrary size $\sigma \leq p$ (where p as usual is the page size). Identical results hold for the LEAST-FREE and MRU strategies. We neglect the space occupied by page headers and left headers since this depends on the implementation of the strategy and in any case amounts to only a small percentage of the total space (see Section 5.3).

To conform with the notation of Robson [14], we shall assume that the maximum block size is n bytes (in the notation of Section 5, $n = s_{max} \leq p$), and that at most M bytes of (virtual or real) memory are requested at any time. In other words, if blocks of sizes $\sigma_1, \ldots, \sigma_m$ have been requested but not freed, then $\sigma_1 + \ldots + \sigma_m \leq M$. For any storage allocation strategy, the memory size S necessary to guarantee that the strategy does not break down due to fragmentation is a function of n and M. Robson [13] has shown that S lies between $\frac{1}{2}(M \log_2 n)$ and about $0.84(M \log_2 n)$ for the optimal strategy.

We shall show that $S \lesssim 1.8842(M \log_2 n)$ for the MOST-FREE strategy, provided that SIZEMAP is chosen appropriately and FREETOL has its minimum value of 1. In other words, the MOST-FREE strategy is within a small constant factor of optimal, so far as its worst-case fragmentation is concerned. Similar results are known for the FIRST-FIT and (binary) BUDDY strategies: Robson [14] has shown that $S \lesssim M \log_2 n$ for FIRST-FIT, and Knuth [10] has shown that $S \lesssim 2(M \log_2 n)$ for BUDDY. Note that the BEST-FIT strategy is much worse: Robson [14] has shown that it has $S \simeq Mn$.

Practical experience and simulations show that the worst case behaviour is extremely unlikely to occur, so the results of this Section have little practical relevance. For more practical results, see Section 7.

6.1 The choice of SIZEMAP

In order to prove Theorem 6.1 we take a particular choice of SIZEMAP. Let $\rho > 1$ be a constant whose optimal value will be determined below. Take $s_1 > 1/(\rho-1)$ and $s_{j+1} = \min(n, \lfloor \rho s_j \rfloor)$ for $j = 1,2,\ldots k-1$, where k is the first index such that $s_k = n$. Then (as in Section 5) $\mathrm{SIZEMAP}[s] = s_1$ if $1 \leq s \leq s_1$, and $\mathrm{SIZEMAP}[s] = s_j$ if $s_{j-1} < s \leq s_j$. We shall assume that requested block sizes lie in the interval $[s_1, s_k]$. For convenience in the statement of the results below we define $s_0 = s_1 - 1$.

6.2 The worst-case bound

Since the proof of the worst-case bound is rather technical, we first state the main result (Theorem 6.1) and then give some Lemmas which are needed for the proof.

THEOREM 6.1:

If FREETOL = 1 and SIZEMAP is as in Section 6.1 with $\rho = e$, then for the MOST-FREE strategy

$$S \leq 1.8842(M \log_2 n) + O(M + p \log n) ,$$

where the constant $1.88\ldots = e.\ln 2$.

LEMMA 6.1:

If $0 < x \leq 1$ then $\dfrac{1}{x\lfloor 1/x \rfloor} < 1 + 2x$.

Proof:

If $x \leq \frac{1}{2}$ then $\frac{1}{x\lfloor 1/x \rfloor} < \frac{1}{x(1/x-1)} = \frac{1}{1-x} \leq 1+2x$. On the other hand, if $\frac{1}{2} < x \leq 1$ then $\frac{1}{x\lfloor 1/x \rfloor} = \frac{1}{x} < 2 < 1+2x$.

LEMMA 6.2:

Those blocks with sizes σ in the interval $(s_j, s_{j+1}]$ occupy at most $1 + \frac{\rho M}{p}\left(1 + \frac{2s_{j+1}}{p}\right)$ pages, for $0 \leq j < k$.

Proof:

There are at most M/s_j such blocks, and $\lfloor p/s_{j+1} \rfloor$ of them fit on one page, so the number of pages required is at most $1 + \frac{M}{s_j \lfloor p/s_{j+1} \rfloor}$. Now apply Lemma 6.1 with $x = s_{j+1}/p$, using the fact that $s_{j+1} \leq \rho s_j$ for $j \geq 1$ (a trivial modification to the argument is required when $j = 0$).

LEMMA 6.3

If k and $s_1, \ldots, s_k$ are defined as in Section 6.1, then

$$k = \log_\rho n + O(1)$$

and

$$\sum_{j=1}^{k} s_j = O(n) .$$

Proof:

From the definition of s_{j+1} in Section 6.1 it is easy to prove that

$$\rho^j \left(s_1 - \frac{1}{\rho - 1}\right) \leq s_{j+1} \leq \rho^j s_1 \quad \text{for} \quad 0 \leq j < k ,$$

and the Lemma follows easily from this.

LEMMA 6.4

$$S \leq (\rho \log_\rho n)M + O(M + p \log n) .$$

Proof:

Sum the bound of Lemma 6.2 for $j = 0,1,\ldots,k-1$, using Lemma 6.3.

Proof of Theorem 6.1:

Simply choose $\rho = e$ in Lemma 6.4 to minimize the factor $\frac{\rho}{\log \rho}$ which multiplies $(M \log n)$ in the dominant term.

Remark:

SIZEMAP was chosen in Section 6.1 to give the smallest possible constant in Theorem 6.1. However, the reader will see from the proof of Theorem 6.1 that rather weak restrictions on SIZEMAP are sufficient to guarantee that the worst case fragmentation for MOST-FREE is within a constant factor of the best possible. Similarly for LEAST-FREE and MRU.

7. EXPERIMENTAL RESULTS FOR BLOCKS OF VARIOUS SIZES

Many programs which use dynamic storage allocation actually request blocks of only a small number of distinct sizes, e.g. for the nodes in certain list structures. For such programs we expect the results of Section 3 to be relevant. However, in other applications a large number of different block sizes are requested, and it is not clear how well we can extrapolate results obtained for a single block size.

A difficulty in comparing dynamic storage allocation strategies is that some assumptions have to be made about the distribution of block size requests and block lifetimes. If we were to compare strategies for artificially simple distributions (e.g. uniform block sizes in some interval and independent uniform or exponential lifetimes) then our conclusions would be unconvincing because of the artificiality of the assumptions. In this Section we use "real" block size and lifetime distributions, although some simplifying assumptions (e.g. regarding independence) still have to be made. Also, in this Section we include the implementation overheads of the different strategies in the comparisons (i.e. we are comparing algorithms which implement the strategies) and we quote actual CPU times and page faults (instead of simulated page faults as in Section 3).

7.1 The CAMBRIDG and YKTVMV distributions

Bozman et al [4] obtained several empirical request distributions by collecting data on requests made to the operating system storage manager in some large time-shared computer systems. For each possible block size s, Bozman et al [4] estimated the mean *inter-arrival time* I_s and *holding time* H_s. They then made the simplifying assumption that requests were independent and exponentially distributed with the empirically observed means I_s and H_s. In other words, they assumed that

a) for each size s, the time between requests for blocks of size s is exponentially distributed with mean I_s, and

b) each request for allocation of a block of size s is followed, after a time which is exponentially distributed with mean H_s, by a request to free the block. For a justification of these simplifying assumptions, see [4].

In this Section we use two of the distributions given by Bozman et al [4]: CAMBRIDG (from an IBM 370/158 UP serving 40-50 simultaneous users at the IBM Cambridge Scientific Center in Cambridge, Massachusetts), and YKTVMV (from an IBM 3033 MP serving 450-540 simultaneous users at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York). For these distributions all block sizes are multiples of 8 bytes in the interval [8,4096]. Some statistics on the distributions are given in Table 7.1. Note that the YKTVMV distribution has about 5 times as many requests per second and more than

10 times as many active blocks as the CAMBRIDG distribution.

Table 7.1: Statistics on the CAMBRIDG and YKTVMV distributions.

		CAMBRIDG	YKTVMV
Mean requests per second	$= \Sigma 1/I_s$	196	1033
mean number of active blocks at equilibrium	$= \Sigma H_s/I_s$	2347	27360
mean storage requested (in bytes)	$= \Sigma s H_s/I_s$	307397	3226427
mean size of a request (in bytes)	$= (\Sigma s/I_s)/(\Sigma 1/I_s)$	98	129
mean block lifetime (in seconds)	$= (\Sigma H_s/I_s)/(\Sigma 1/I_s)$	12	26

7.2 Comparison of storage efficiencies and items visited

In their comparisons of different dynamic storage allocation algorithms, Bozman et al [4] considered *storage efficiency* and *items visited.* The latter is the number of items visited on linked lists or added to linked lists (with items added to doubly linked lists given double weight), and is intended to provide a rough measure of the CPU time requirements of the algorithms in a machine-independent manner. In Table 7.2 we give the mean storage efficiencies and mean items visited (per request for a block and its subsequent freeing) for several algorithms and both the CAMBRIDG and YKTVMV distributions. We give results for two different page sizes:

$p = 512$ bytes (as on the VAX 11/750 and as in Section 3);

$p = 4096$ bytes (as in Bozman et al [4]).

In obtaining our results we ignored blocks of size s with $H_s \leq 0.01$ sec ; these would have made a negligible difference.

Table 7.2: Comparison of storage efficiencies and items visited.

Strategy	Page size	FFCROSSOVER	Mean storage efficiency		Mean items visited	
			CAMBRIDG	YKTVMV	CAMBRIDG	YKTVMV
Standard IBM (1)	4096		0.843	0.887	9.7	108.7
First-fit (1)			0.844	0.926	425.4	3270.0
First-fit (2)	512	0	0.844	0.877	20.58	20.14
	4096	0	0.852	0.902	107.94	115.19
Next-fit (1,3)			0.570	0.567	207.4	1892.8
Binary buddy (1,4)			0.683	0.641	5.01	5.00
Binary buddy (1,5)			0.688	0.716	16.31	50.44
LEAST-FREE (6)	512	120	0.830	0.886	5.67	5.82
	512	512	0.759	0.811	5.45	5.33
	4096	120	0.806	0.913	6.31	8.50
	4096	4096	0.750	0.896	5.05	5.03

Notes

(1) As given in Table 3 of Bozman et al [4].

(2) Implemented as in Brent [5] with segment size = page size.

(3) "Next-fit" is Bozman et al's "mod-first-fit": see Hext [7] and Knuth [10], exercise 2.5.6.

(4) Binary buddy [8,9,10,11] with tag bits.

(5) Binary buddy [8,9,10,11] without tag bits.

(6) LEAST-FREE implemented as described in Sections 4-5, with FREETOL = 1, SIZEMAP chosen as described at the end of Section 5, and FIRST-FIT used for blocks of size exceeding FFCROSSOVER. Means are over 4000 seconds of simulated time. Results for the MOST-FREE and MRU strategies are similar to those for LEAST-FREE.

From Table 7.2 we see that for some algorithms the mean items visited (per block requested and released) increases almost linearly as the number of blocks allocated increases. This is true for the standard IBM algorithm, the first-fit algorithm implemented in the obvious way, and the next-fit algorithm. Hence, these algorithms become impractical because of their CPU time requirements if a large number of blocks are allocated. In contrast, the mean items visited is almost independent of the number of blocks allocated for the LEAST-FREE algorithm, the binary buddy algorithm (with tag bits), and the first-fit algorithm implemented as in Brent [5]. Of these binary buddy (with tag bits) and LEAST-FREE visit less than 10 items per block requested and freed. However, the binary buddy algorithm gives significantly lower storage efficiency than LEAST-FREE, so LEAST-FREE is to be preferred.

The choice of FFCROSSOVER is a trade-off between storage efficiency and CPU time (or items visited). With FFCROSSOVER = p we have almost "pure" LEAST-FREE with 5-6 items visited per request. With FFCROSSOVER = 0 we have the first fit algorithm with a considerably larger number (depending on the segment size) of items visited per request, but significantly higher storage efficiency, especially for CAMBRIDG. The compromise of FFCROSSOVER = 120 seems to come close to obtaining both high storage efficiency and low items visited per request.

The use of LEAST-FREE for blocks of size $s \le$ FFCROSSOVER in order to improve execution time of FIRST-FIT is similar to Bozman's use [3] of "software lookaside buffers". As far as storage efficiency and items visited are concerned, the LEAST-FREE algorithm is very similar to the "subpooling" algorithms of Bozman et al [4]. However, we would expect their paging behaviour to differ because the subpooling algorithms essentially use the STACK strategy in each subpool (see Sections 2-3).

7.3 Comparison of paging behaviour of different algorithms

Bozman et al [4] did not obtain any data on how often each block was referenced. Hence, to compare the paging behaviour of different dynamic storage allocation algorithms with the CAMBRIDG and YKTVMV distributions, we simply assumed that each block was referenced twice, once when allocated and once just before being freed. This assumption certainly underestimates the number of references to each block, but there is no obvious reason why it should favour one of the dynamic storage allocation algorithms over the others.

In Table 7.3 we give the (real) page faults and CPU time required on a VAX 11/750 computer when simulating the YKTVMV distribution for $t = 2000$ (simulated) seconds with a working set of 3000 512-byte pages. The expected inter-arrival times I_s given in [4] were divided by a "load factor" L so that results could be obtained for different loads (the expected number of blocks allocated is 27360L and the expected amount of storage requested is 3226427L bytes). In Table 7.3, "E" is the mean storage efficiency, "pf/t" is the number of page faults per simulated second, and "CPU/t" is the CPU time (in seconds) per simulated second. The CPU time required for random number generation and simulation is included, but is not very significant. A fast time-indexed simulation algorithm [19] was used rather than the leftist tree algorithm of Section 3.

In addition to results for the FIRST-FIT, LEAST-FREE, MOST-FREE, MRU and QUICK algorithms, we include in Table 7.3 some results obtained using the standard VAX VMS library routines LIB$GET_VM and LIB$FREE_VM (which are called by Pascal "new" and "dispose" procedures with additional space and time overheads).

Table 7.3: Comparison of dynamic storage algorithms for YKTVMV distribution.

Algorithm	FFCROSSOVER	Load factor 0.50			Load factor 0.75			Load factor 1.00		
		E	pf/t	CPU/t	E	pf/t	CPU/t	E	pf/t	CPU/t
QUICK	120	0.351	80.2	1.319	0.355	276.0	2.194	0.352	533.4	3.227
	508	0.300	70.2	1.281	0.301	188.9	2.034	0.304	419.3	2.943
FIRST-FIT (1)	0	0.877	22.5	1.770	0.879	131.0	2.866	0.891	396.7	4.124
MOST-FREE (2)	120	0.881	20.5	1.394	0.893	71.5	2.120	0.890	221.9	3.031
	508	0.786	18.5	1.313	0.789	49.2	2.027	0.793	163.8	2.831
LEAST-FREE (2)	120	0.886	17.5	1.400	0.900	76.0	2.139	0.894	213.1	3.039
	508	0.792	14.8	1.365	0.796	45.2	2.101	0.798	157.6	2.872
MRU (2)	120	0.884	18.9	1.363	0.901	74.5	2.081	0.892	226.3	2.954
	508	0.783	18.8	1.342	0.788	54.5	1.996	0.791	160.6	2.801
VAX-VMS (3)	-	-	156.6	12.048	-	1121.6	24.941	-	33557.9	60.562

Notes

(1) Implemented as in Brent [5] with segment size 512 bytes and segments aligned on page boundaries.

(2) Implemented as described in Sections 4-5 with FREETOL = 1. Note that 508 is the maximum allowable value of FFCROSSOVER since each page contains a 4-byte pointer back to its page header.

(3) Using VAX VMS library routines LIB$GET_VM and LIB$FREE_VM. Storage efficiency E not determined, but probably slightly lower than for our FIRST-FIT algorithm.

From Table 7.3 we see that the QUICK strategy is indeed fast, although not significantly faster than the MRU strategy. MRU is preferable because QUICK generates more page faults and gives low storage efficiency.

We were unable to obtain comparable results for NOFREE because of paging file overflow. However, runs with $t < 2000$ showed that NOFREE performs worse than QUICK.

The FIRST-FIT strategy, implemented as in Brent [5], gives good storage efficiency and reasonable CPU times, but it is significantly slower than the MRU strategy. The results for FIRST-FIT and VAX-VMS show the difference between two implementations of the first fit strategy. A curious observation is that for VAX-VMS the entries pf/t and CPU/t increase with the simulated time t (e.g. for load factor 1.00 and t = 4000 we have pf/t = 87777 and CPU/t = 91.561; for t = 2000 as in Table 7.3 we have pf/t = 33557 and CPU/t = 60.562). For the other algorithms pf/t and CPU/t are almost independent of t .

The behaviour of MOST-FREE, LEAST-FREE and MRU is very similar. For all three strategies the page faults and CPU time are decreased by choosing the maximal value of FFCROSSOVER (i.e. 508) instead of 120, but this is at the expense of lowering the storage efficiency by about 10 percent. If FFCROSSOVER is reduced below 120 the behaviour of the strategies becomes close to that of FIRST-FIT (which is the limiting case FFCROSSOVER = 0), i.e. the storage efficiency stays about the same but page faults and CPU time increase.

The results given in Table 7.3 are all for FREETOL = 1. We tried larger values of FREETOL, but the results were generally worse than with FREETOL = 1 (compare the results of Section 3.4).

8. CONCLUSION

The criteria which should be used to compare dynamic storage allocation strategies (and 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 three new strategies (the MOST-FREE, LEAST-FREE and MRU strategies) which are intended to be good on a computer with virtual memory. The new strategies can be implemented efficiently (as discussed in Sections 4-5) and have good worst-case behaviour (as discussed in Section 6). The new strategies compare well with traditional strategies, both in the general case (allocation of blocks of various sizes) and in the special case that all blocks have the same size: see Sections 3 and 7 for experimental results. The difference between the three new strategies is small, but overall the MRU strategy may perform slightly better than the MOST-FREE and LEAST-FREE strategies. Even on a computer without virtual memory, the MRU strategy may be a good choice because it can be implemented efficiently and gives high storage efficiency.

Because the new strategies are only applicable for blocks of size smaller than the page size, they have to be combined with another strategy which is used for large blocks. To obtain the results quoted in Section 7 we combined the new strategies with an efficient implementation of the FIRST-FIT strategy.

As mentioned in Section 2.12, 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.

All the storage allocation strategies discussed in this paper have been implemented in (slightly machine-dependent) Pascal on a VAX computer running under VMS: readers interested in obtaining further information should contact the author.

REFERENCES

1. BAYS, C., A comparison of next-fit, first-fit and best-fit, *Comm. ACM* 20, 3 (March 1977), 191-192.

2. BOJANCZYK, A. and BRENT, R.P., Tridiagonalization of a symmetric matrix on a square array of mesh-connected processors, Report CMA-R45-83, Centre for Mathematical Analysis, Australian National University, Dec. 1983.

3. BOZMAN, G., The software lookaside buffer reduces search overhead with linked lists, *Comm. ACM* 27, 3 (March 1984), 222-227.

4. BOZMAN, G., BUCO, W., DALY, T.P. and TETZLAFF, W.H., Analysis of free-storage algorithms - revisited, *IBM Systems Journal* 23, 1 (1984), 44-64.

5. BRENT, R.P., Efficient implementation of the first-fit strategy for dynamic storage allocation, submitted to *ACM Trans. on Programming Languages and Systems*. Available as Report CMA-R33-84, Centre for Mathematical Analysis, Australian National University, August 1984.

6. DENNING, P.J., Virtual memory, *Computing Surveys* 2, 3 (Sept. 1970), 153-189.

7. HEXT, J.B. A storage management laboratory, *Australian Computer Science Communications* 2, 1 (Jan. 1980), 185-193.

8. KAUFMAN, A. Tailored-list and recombination-delaying buddy systems, *ACM Trans. on Programming Languages and Systems* 6, 1 (Jan. 1984), 118-125.

9. KNOWLTON, K.C., A fast storage allocator, *Comm. ACM* 8, 10 (Oct. 1965), 623-625.

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

11. PETERSON, J.L. and NORMAN, T.A., Buddy systems, *Comm. ACM* 20, 6 (June 1977), 421-431.

12. REEVES, C.M., Free store distribution under random fit allocation: Part 2, *Computer Journal* 23, 4 (1980), 298-306.

13. ROBSON, J.M. Bounds for some functions concerning dynamic storage allocation, *J.ACM* 21 (1974), 491-499.

14. ROBSON, J.M., Worst case fragmentation of first fit and best fit storage sllocation strategies, *Computer Journal* 20, 3 (1977), 242-244.

15. SHORE, J.E., On the external storage fragmentation produced by first-fit and best-fit allocation strategies, *Comm. ACM* 18, 8 (Aug. 1975), 433-440.

16. SHORE, J.E., Anomalous behaviour of the fifty-percent rule in dynamic memory allocation, *Comm. ACM* 20, 11 (Nov. 1977), 812-820.

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

18. STEPHENSON, C.J. 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*.

19. WYMAN, P.F., Improved event-scanning mechanisms for discrete event simulations, *Comm. ACM* 18, 6 (June 1975), 350-353.