A dynamic disc allocation algorithm designed to reduce fragmentation during file reloading

B. J. Austin

Information Sciences Laboratory, Corporate Research and Development Center, General Electric Company, Schenectady, NY, USA

This paper describes an algorithm for allocating file storage on a disc. An attempt is made to allocate space for a file in contiguous disc addresses. Failure to find such contiguous storage results in formation of a 'page table', so that non-contiguous physical addresses appear contiguous in the logical address space presented to the programmer. Since an extra disc operation may be necessary to access a file with a page table, it is desirable to minimise the number of files with page tables, and maximise the chance of finding contiguous storage. This paper describes an attempt to achieve these ends and gives some measurements of its performance. The method uses daily dumping and reloading of the file storage. The allocation procedure has been tuned to optimum performance during reloading, but it continues to function satisfactorily during the remainder of the day.

(Received October 1970)

1. Introduction

This paper describes an algorithm for disc allocation used in an operating system written at the Research and Development Center. Since the operating system has been described elsewhere (Kerr, Bernstein, Detlefsen, and Johnston, 1969), (Bernstein and Hamm, 1969) and (Bernstein, Detlefsen, and Kerr, 1969) no details will be given here. Suffice it to say that the system allows the multiprogramming of a number of processes on a modified General Electric 600 computer, and has at the time of writing 16M words of non-removable file storage on a DSU10F Disc Storage Unit. For the purposes of allocation this space is divided into 32K 'pages' currently of 512 words. For the remainder of the paper it will be more convenient to speak in terms of pages and these units will be adhered to as far as possible.

A file presents to the programmer a logical addressing space consisting of consecutively numbered disc records, starting at 0. Wherever possible the system allocates this space to contiguous physical addresses, but should this be impossible a 'page table' is constructed. A page table is simply an array of page addresses giving the mapping between the logical file addresses and physical disc addresses.

This situation is undesirable in two ways. Firstly, page tables are held on the disc, and therefore an extra disc access may be required for a file with non-contiguous allocation. The word 'may' is used since active page tables are held in a set of core buffers. However, when the number of currently active files having page tables is large, the page table traffic is high and system performance drops. Secondly, the non-contiguity of disc allocation implies that a single request in the programmer's address space may have to be serviced as a number of separate disc operations.

Clearly, any technique which reduces fragmentation of the disc space and minimises the number of page tables is desirable. This paper attempts to set forth such a technique, which specifically is aimed at reducing the amount of non-contiguous allocation and making best use of the disc hardware. To put this technique in perspective, a description of the previously-used algorithm and its performance will be given.

Prior to the work described in this paper, the disc allocation was managed by a method due to J. L. Smith. Disc allocation was administered by means of a map held in core, with one bit per disc page. The method to be described in this paper retains all of Smith's procedures, except that the technique for searching the map has been improved. An allocation request was satisfied, if possible, by finding a suitable number of adjacent pages. Allocation was dynamic in the sense that space was given to a file only as needed, not being reserved in advance. The system also maintained a list of 'holes' in the disc map. This list was added to whenever a search for allocation found a string of adjacent free pages whose length was less than required for the request in hand. The list was regarded as secondary to the bit map, however, and recourse was made to the map to check the existence of an alleged area of free space. Although it represents a worthwhile economy, the table of available strings of pages will be ignored in this paper.

If a file expanded, additional space was sought first in pages contiguous to the existing allocation. When this was impossible, a page table was necessary. Note that no attempt was made to relocate the existing part of the file in an attempt to obtain a larger free area.

The bit map was divided into two equal sections and each section had a pointer showing where the next map search would begin. Essentially the first map section was used for satisfying small allocation requests (1 page) and the second section for larger requests. When any allocation was made, the associated pointer was incremented by a small number of pages (2 in section 1, 7 in section 2) to allow for later expansion of the file. This space was not reserved for the file, but the intention was that growth of the file could be accommodated in it. The algorithm performed quite well when the disc had a large amount of available space, (say 50% of the total) but, unfortunately, when the disc became full, the map pointers would sweep over their respective sections many times, so that the map became very badly fragmented with large numbers of small holes. Furthermore, second and subsequent sweeps of the pointers would cause the areas reserved for file growth to be used, thereby blocking any expansion of previously allocated space.

As standard operational practice, we have been in the habit of performing a logical dump onto magnetic tape of the entire file structure once per day. The dump program is a normal (slave) user process, and can be run concurrently with normal system use. The dump is intended as a backup procedure, since our disc storage is non-removable. We have also usually reloaded these dump tapes once per day, and again the load program runs as a normal slave process. Generally, we reload the file storage by multiprogramming several copies of the load program. The reloading is intended to achieve two objectives. Firstly, a system crash could cause loss of some disc space, i.e. that allocated to non-catalogued files at the time of the failure. We have now overcome this problem by means of a salvage program which checks the directory structure, the page tables, and the disc allocation map for consistency and correctness, reconstructing the map to show only space allocated to catalogued files. Secondly, the reloading was intended to compact

Table 1 Distribution of file sizes

FILE SIZE (PAGES)	NUMBER OF FILES	CUMULATIVE ALLOCATION
1	2,460	2,460
2 to 3	1,388	5,750
4 to 7	779	9,729
8 to 15	489	14,871
16 to 31	217	19,406
32 to 63	61	22,006
64 to 127	36	24,912
128 to 255	9	26,538
256 to 511	0	26,538
512 to 1023	4	29,080

Total number of files = 5,443

Percentage of available space used = 88·7
Percentage of space used by files of less than 16 pages = 45·4

the storage, so that the system would at least start the day with a small number of page tables. Unfortunately, this second objective was not achieved, and while the fragmentation did get worse as the day progressed, it was already very bad immediately after the file load. Reference to **Tables 1 and 2** will show that out of about 5,400 files existing at the time of the measurement, 540 required a page table during a fresh loading of the disc. At times when the disc was more fully occupied the fraction of files with page tables approached 20%.

It was also observed that when the disc space was very fully used (say 96 to 98%) a perceptible drop in system performance occurred due to time spent by the executive in abortive scans of the disc map. A worse case scan, in which the whole map consisted of alternate ones and zeros, could take some hundreds of milliseconds. This is clearly intolerable, and the searching algorithm should attempt to reduce fragmentation for this reason alone, quite apart from page table considerations.

2. The new algorithm

The method just described suffers from the following weaknesses:

1. Allocation is made on the basis of the current request, rather than on the expected file size.
2. Space is not allocated at an equal rate within the two map sections. This results in section 2 becoming full first, and considerable processor time is used in abortive scans of large fractions of the map.
3. The concept of allowing space for possible file growth does not work when the disc approaches full utilisation, and merely results in the fragmentation of available storage into many small (and not very useful) pieces.

Various proposals were examined with a view to improving this situation. Firstly, the possibility of static allocation was explored. This would imply the reservation of a contiguous area sufficient to cover all of a file's expected growth. When a file is first opened, the system requires, among other parameters, a statement of the maximum length to which the file will be allowed to grow. This maximum length could therefore be used to set up a static allocation. Unfortunately, most files are opened on the user's behalf by some subsystem, e.g. TSS, the Time Sharing System, and the subsystem can only supply an approximate upper bound on the file length. Furthermore, it would be unreasonable to expect the user to supply any meaningful limit. This proposal was therefore abandoned.

Table 2 File loading experiment

	PREVIOUS ALLOCATION ALGORITHM	NEW ALLOCATION ALGORITHM
Time for load (minutes)	64	48
Files of 1 to 63 pages with PT	493	41
Files of 64 to 127 pages with PT	35	29
Files of 128 to 255 pages with PT	8	6
Files of 256 or more pages with PT	4	4
Total page tables	540	80

A second and more fruitful idea was to concentrate on loading the file structure well, and let the rest of the day take care of itself. The file loading program could explicitly give the disc allocation routine the length of the incoming file, rather than merely implying its length by writing a series of buffers to the disc. As has been mentioned above, the file loading program is a normal user process and it is treated exactly as any other process for the purposes of disc allocation. There is a special privilege granted to the file loader in the directory handler, and in principle a primitive could have been added to the system to allow privileged access to the disc allocation. Such a primitive might even have been made generally available, and would have allowed virtually perfect compaction of the disc after a reloading operation. However, a substantial amount of effort would be required to add such a primitive to the system, and it was decided to make a minimum modification to the system and to alter the (more tractable) file loading program to cooperate with the disc allocation routines to give a well compacted disc. The modified algorithm which was finally arrived at applies both to the file loading program and to normal system use, and is as follows.

The DSU10F Disc Storage Unit has a maximum allowable transfer size of 16 pages (i.e. 8K words). Thus, without regard to map fragmentation, but merely from an efficiency standpoint, it was decided that files should be written to the disc in units of up to 16 pages. Also, reference to Table 1 will show that about half of the disc space occupied by permanent files belongs to files of less than 16 pages. Therefore, the disc map is divided into two equal sections, which are essentially managed by different strategies—section 1 has a strategy for files up to 15 pages, while section 2 has a strategy for larger files. The file loading program reads records from tape until either the whole file is in core or a buffer of 8K words is filled. The information is then written to the disc. The disc allocation therefore has either the exact file length (for small files) or the knowledge that it is a large file (i.e. 16 or more pages).

A small file is written as a single unit and therefore all its space is allocated at one time. The allocation is attempted first in section 1, but will be made in section 2 if that fails. When an allocation is made in the first section the map search pointer is incremented by 16 pages. This is not an allowance for file growth, but a means whereby the first pass over the section leaves a population of holes suitable for satisfying subsequent requests in that map section.

Large files are written to the disc in a series of operations, of which the first is 16 pages long. Allocation is sought for the first operation in section 2, and thereafter contiguously. The map pointer is incremented to allow a maximum contiguous file size of 64 pages. Thus, it is approximately true that files up to 64 pages should not have a page table, whereas files over 64 pages definitely will. The approximation is due to two causes, viz:

1. On the first pass through the map, an area of more than 64 contiguous pages may be allocated.
2. On subsequent passes, the maximum sized hole is 64—16 pages.

The increment 64 has not been chosen to produce a minimum

number of page tables. Indeed, some simulation experiments showed that the number of page tables could be reduced almost to zero with larger increments (up to 1,024). However, it was felt that the reduction expected with an increment of 64 would be adequate and other factors such as optimum placement of files with respect to physical barriers of the DSU10F disc and fragmentation due to file activity during the day's operations would be made worse by increasing the increment.

A crucial addition to the allocation algorithm has not yet been mentioned. In section 1 (2), after the map pointer is incremented by 16(64) it is rounded down to a multiple of 16(64). In other words, the pointer is advanced after allocation to the next multiple of 16(64). This device has a marked effect on reducing fragmentation on second and subsequent passes through the map section. It ensures that when the next search is begun, the first free page found will be on the edge of a free hole and not in the middle. It does imply that a few bits will be scanned fruitlessly, but the effect on fragmentation is quite dramatic.

To conclude this section, a summary of the effects of the allocation procedure will be given. Firstly, files less than 16 pages (8K words) have a very high probability of obtaining contiguous allocation. Furthermore, such files should never cross the barrier which exists at every multiple of 8K words (a property peculiar to the DSU10F), and therefore a very large fraction of our permanent files should be accessible in one disc operation. Files between 16 and 63 pages should also get contiguous allocation, but with much less certainty. Files of 64 or more pages are very likely to have a page table.

In more general terms, the design sequence uses the available degrees of freedom as follows:

1. The transfer size for the loading program is chosen from a consideration of the physical properties of the DSU10F disc. Let this be denoted by S.
2. S defines the boundary between 'small' and 'large' files. The map is divided into two sections to be used for small and large files respectively.
3. The parameters of the strategy for handling small files are also dependent on S. A small file is of up to S-1 pages. A search for free space always begins at a multiple of S, and therefore the first pass through this map section leaves a population of holes from 1 page to S-1 pages.
4. The parameters of the strategy for handling large files are chosen by experiment. A search for free space starts at a multiple of some constant T, and therefore one can generally expect a page table for a file of over T pages. By increasing T sufficiently, the number of page tables in the file structure after a complete load can be made very small. There is, however, a diminishing returns situation—the system can tolerate a few page tables—and T is set at a value which requires about 1 to 2% of the files to have non-contiguous allocation.

3. Simulations and measurements

As an aid to an understanding of the allocation process and in the devising of an algorithm to reduce fragmentation, a model of the file loading procedure was constructed. Since a dump of the complete file structure occupies many tapes, it is customary to run several copies of the loading program concurrently. It can be seen that competition among these processes for disc allocation could lead to fragmentation. A crude model of the concurrent running of four loading programs was constructed. It was assumed that they would all fit in core and would be I/O limited. Thus a detailed model of the swapping and timeslicing aspects of the system scheduler was unnecessary. Furthermore, fixed times were assumed for disc and drum (i.e. directory) operations. The model was furnished with the file lengths from a set of actual dump tapes (the same files which gave Table 1). With the allocation algorithm described above, the model predicted 55 files requiring page tables. Note, from Table 1, that there are 49 files of 64 or more pages.

Experiments were performed with the operating system using the same dump tapes. As Table 2 shows, the number of page tables required was 80. It should be mentioned that in our system a programmer may produce a file in which the *logical* addressing space is non-contiguous. Such a file will always have a page table. It is thought that most of the difference between the results of the model and actual experiment is due to such files. Table 2 also shows that the time for the loading process has been considerably reduced. This is due to two causes, viz:

1. The number of disc operations is reduced.
2. The time spent searching the map is reduced.

The dumping and reloading of the file storage takes a little less than two hours of machine time—the load takes about three quarters of an hour and the dump, which involves verification of the tapes produced, takes about an hour. The cost of the compaction produced by the allocation algorithm of this paper should be counted as only the time to load the files since the daily dump is performed for security. It should also be noted that the dump/load procedure is performed overnight, i.e. not during prime time.

Since the algorithm has been adjusted for the file loading process, but also applies during normal operation, it is worth discussing its behaviour in everyday use. The number of page tables at the beginning of the day has been consistently less than 100, and during a day's activities the disc becomes fragmented to the extent that about 100 more page tables are formed. These remarks apply to a period during which the disc was between 88 and 98% full. With the previous algorithm in similar circumstances the number of page tables immediately after a file load was 500 to 1,000, and during the day an additional 100 to 200 would be formed. It should be emphasised that these numbers have much lower dependability than the results shown in Table 2, since they are the product of normal day-to-day system use rather than of a controlled reproducible experiment.

The algorithm has made a significant improvement in the file loading time, but this is relatively unimportant compared to the effect on normal computer operation. The fraction of the disc traffic concerned with page table transfers has dropped from about 10% to about 1%. The system no longer spends a large amount of time searching the allocation map when the disc is practically full, but unfortunately no quantitative evaluation of this effect is available.

It can be said, however, that the algorithm of this paper seems satisfactory for both file loading and normal operation. This must mean that the amount of file activity which causes file growth is fairly low. Unfortunately, while we have a monitoring of accesses to every file, we have no means at present of telling what percentage of our files is changed each day. However, of a sample of the files which made up Table 1 only 10% had been accessed in the last day. Presumably then a much smaller fraction of the files are modified in such a way as to require a larger number of pages.

An unexpected benefit of the new scheme has been an apparent increase in the file system reliability. In the past, the code concerned with handling page tables has been found to contain many subtle errors, whereas the code which deals with contiguous files has proved relatively troublefree. There may well be a number of errors still in the page table code but the probability of achieving the proper circumstances to trigger them has been reduced drastically.

A failing that has been observed is that temporary files which exist during the day, and which were not taken into account when 'tuning' the algorithm, can upset the balance of storage between the two map sections. In other words, unequal map sections should have been set up, so that the allocation scheme

would work for the combination of permanent plus temporary files. Some of these temporary files also have the unfortunate property of expanding to quite large size by single page increments. Thus the fact that the size of the current request rather than the expected file length determines the map section shows up as a fundamental weakness. It is intended to improve the system so that some use is made of the expected file length, while not requiring that a user or subsystem supply an exact estimate.

4. Discussion

It will be clear that the allocation method described in this paper has been very carefully adjusted to both the characteristics of the DSU10F disc and the particular operating environment which exists currently in our system. This section will attempt to bring out some points of more general applicability.

Firstly, it may be remembered that the real test of an allocation algorithm does not come until the disc space is nearly full, and that almost any algorithm will handle the situation when space is plentiful. Indeed, in one operating system known to the author, it appears that additional disc units are purchased as necessary to keep the allocation scheme working. The allocation pointer moves monotonically through the available space, leaving some wasted pages so that files may expand. If the pointer reaches the end of the last disc unit, a compaction is required. The number of disc units is adjusted so that compaction need not be unreasonably frequent (e.g. more than once per week).

A second point is that it is very difficult to set up an efficient algorithm at the time of system design. The parameters of any algorithm cannot be set for optimum performance until the system has been in use for some time so that a large amount of disc space is in use, and the equilibrium between user practice and the administrative purging policy has been established. Furthermore, it may be necessary to retune the algorithm for every installation and even at one installation if more disc storage is acquired. It would appear that an allocation procedure which automatically adapts to the equilibrium file distribution is desirable, but no such scheme is known to the author.

Other operating systems have employed various techniques for holding the record of available space and for representing the space allocated to a file. The DAD system (Austin, Holden, and Hudson (1967)) uses a bit map for available space and employs what Denning (1967) rightly describes as 'the common and questionable practice' of chaining pages together within themselves. This approach makes allocation easy, since all allocation requests are exactly one page long, but makes very small use of contiguity in performing transfers efficiently. (There are also other difficulties, such as weakness in random file operations.)

The General Electric Comprehensive Operating System (GECOS) holds in core the 64 longest strings of available 'links' and represents space allocated to a file as a series of number pairs in the file catalogue showing disc address and length. This approach makes use of contiguity in performing transfers and is economical in searching for allocation. Free space can be lost (temporarily), however, and this system does not seem to make maximum use of contiguity.

It is clear that retention in core of the entire bit map is impractical for a very large disc store, so that the allocation method described in this paper requires some modification. A possible improvement would be to hold only sections of the total map in core at any time. This would, of course, introduce some difficulties but it is felt that the bit map approach has some merit especially in an environment where many users are concurrently accessing the file structure.

5. Acknowledgements

The algorithm presented in this paper owes a great deal to the work of J. L. Smith. In fact, the only change from his version, apart from alterations to parameters, has been the addition of rounding the map search pointer to a multiple of 16 or 64 after every allocation.

The author would like to acknowledge helpful discussions with W. E. Davidsen, R. H. Kerr, G. D. Pearsall, and J. N. Roberts.

References

AUSTIN, B. J., HOLDEN, T. S., and HUDSON, R. H. (1967). DAD, the C.S.I.R.O. operating system. *Communications of the ACM*, Vol. 10, Number 9, September 1967, pp. 575-583.

BERNSTEIN, A. J., DETLEFSEN, G. D., and KERR, R. H. (1969). Process Control and Communication. Proc. Second ACM Symposium on Operating System Principles, Princeton, October 1969, pp. 60-66.

BERNSTEIN, A. J., and HAMM, J. B. (1969). The Design and Implementation of a Directory Hierarchy for a General Purpose Operating System. General Electric Report Number 69-C-356.

DENNING, P. J. (1967). Effects of scheduling on file memory operations. Proc. Spring Joint Computer Conference, 1967, pp. 9-21.

KERR, R. H., BERNSTEIN, A. J., DETLEFSEN, G. D., and JOHNSTON, J. B. (1969). Overview of the R and DC Operating System. General Electric Report Number 69-C-355.

Book review

Approaches to Non-Numerical Problem Solving, by R. B. Banerji and M. D. Mesarovic (editors), 1970; 466 pages. (*Springer-Verlag*, $6·60)

This book is a collection of 18 papers on Artificial Intelligence from a symposium held at Case Western Reserve University in November, 1968. Most of the authors have done widely recognised work in the subject and the book forms a very useful addition to the literature. Although there is little in the way of original research which is not available in the journals, the authors present a synoptic view of a number of topics with references up to 1969. Such overviews include J. A. Robinson on Mechanical Theorem Proving, G. W. Ernst on GPS, R. F. Simmons on Natural Language Question Answering Systems, J. R. Slagle on Heuristic Search Programs, and R. L. London on Proving Correctness of Computer Programs. These papers should be of interest both to the non-specialist who would like to know what has been done in these areas, and to the research worker who would like to stand back and take stock of progress and trends. A. Newell contributes an illuminating, slightly pessimistic, discussion of the relationship between artificial intelligence and cognitive psychology, and B. Raphael explains how the development of robots presents important research problems for the field. Three papers discuss formalisation of problem solving systems and the related question of representation, whilst others outline research on specific projects such as a GO-playing program, a technique for heuristic solution of combinatorial problems, a problem solving programming language, and the impressive Stanford project on computer determination of the structure of organic molecules.

The remaining papers are brief essays by authors mainly distinguished in other fields, including some pithy and lucid remarks by Hao Wang on the interaction between computing and mathematics.

I hope that the rather unappetising title will not deter people working in computing from dipping into this timely and informative volume.

R. M. BURSTALL (Edinburgh)