Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

T. Budd

The cleaning person algorithm

Computer Science/Department of Algorithmics & Architecture Report CS-R8610 February

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

The Cleaning Person Algorithm

Tim Budd

The language B is intended to provide a powerful tool that can nevertheless be used by novice programmers in the solution of nontrivial problems. Part of the design of B involves removing from the programmers concern limitations imposed by such factors as machine word size or memory size. This paper describes an algorithm that permits values to migrate easily between primary and secondary memory (or disk), permitting the B system to act as if the amount of memory was essentially limitless.

1982 CR Categories: E.2, D.3.3, D.3.2
Key Words & Phrases: Programming Language Implementation, Data Types and Structures, *B*

Introduction

The language B is designed to be used by individuals having little or no previous programming experience, working on a personal computer. A primary aim of the language is to provide a simple and easy to understand tool that is nevertheless powerful enough to facilitate the solution of nontrivial problems. To this end, many of the details of the underlying machine that conventionally a programmer must be concerned with can be ignored in B. For example, rational numbers are maintained internally in B in an unbounded form; thus the user does not need to be aware of the machine "word size" or of exceeding the hardware precision of numerical values. In a similar manner, the user should not need to be aware of the limitations imposed by a finite memory on a computer; it should appear to the user as if the amount of memory available is essentially limitless. This paper will describe an algorithm used in achieving this latter goal. We will not present an overview of the B language, good descriptions can be found in [1, 2, 5, 6].

There are several ways in which an executing B system can run out of memory. The most obvious fashion is by creating large objects. Just as there are no intrinsic limitations on the size of numbers, there should be no reason why a user could not create a table containing many thousands of entries, even if the size of the table exceeded available memory. However, memory can also be quickly consumed by a proliferation of small objects. This is particularly true in the B system, since the values of all identifiers are saved at the end of each command. Saving the memory in this manner permits the user to back up execution or to make changes to previous commands. These facilities will be discussed in a later section; for now it is only necessary to note that this feature tends to consume a considerable amount of primary memory that is only infrequently accessed.

In short, a problem can arise where there are too many objects and too little memory. This paper presents one solution to this problem. As the B system is executing, a second process (*the cleaning person*), running in parallel, attempts to move objects from primary memory to secondary storage (such as a disk). Some special characteristics of B objects, notably that once created they are never modified, make this solution feasible.

Report CS-R8610
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

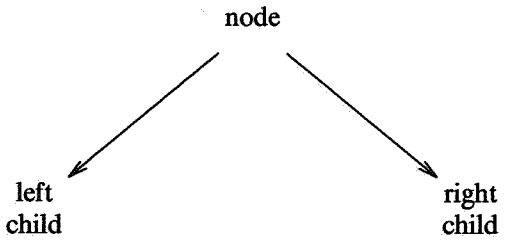

Figure 1: The structure of a node

BACKGROUND

In B, there are two types of basic objects, namely texts and numbers; and three varieties of data structures, namely compounds (heterogeneous collections of fixed size), lists (homogeneous collections of indeterminate size) and tables (homogeneous collections of key/value pairs). Objects of type compound are unsorted, whereas lists and tables are kept in a sorted representation. Currently texts, lists and tables are maintained internally using a clever B-tree representation. (See [4] and [7] for a more complete description of how B is implemented). For the purposes of this paper the fact that B-trees are used, rather than some other tree representation, is immaterial and can be ignored. In the presentation we will consistently deal only with conventional, perhaps unbalanced, binary trees. The algorithm, however, makes no assumptions concerning the type of nodes or the maximum number of children any node can possess. For our purposes, we can assume there are just two types of objects in memory. The first type is a binary tree node, which has two fields (Figure 1), representing pointers to the left and right children, and which may be **nil** if no such child exists. The second type is a leaf node, that is, a node with no children. This could correspond, for example, to a number.

The definition of B [5] uses value semantics for assignment, but the implementation [7] uses pointers. This is possible as follows. Suppose the values denoted by some identifier "*a*" are assigned to an identifier "*b*". The value placed in *b* is merely a pointer to the same object as in *a*. Figure 2 shows this relationship, with the numbers in nodes being used to represent the reference count. If *b* is subsequently modified, then a copy of only the modified portion of the tree is created (Figure 3). Thus *a* and *b* may share common entries; however, *a* still points to the original structure while *b* points to a new structure. These pointer semantics can be applied uniformly, leaving it to the memory manager to discover what nodes can subsequently be recovered. This trick makes assignment and modification of complicated data structures, such as tables and lists, very inexpensive [4].

The global environment is maintained as a B table, the key fields being texts representing names of identifiers and the associate fields maintaining the values for the given identifiers. This fact, combined with pointer semantics, makes it trivial to insure that functions are side-effect free. When a function is called a new copy of the global environment is created (since the copy operation is inexpensive). Any modifications to the environment will then not affect the original. Upon return the copied environment is discarded (automatically, by decrementing its reference count), and the original is thus preserved side effect free.

There are two observations we can make concerning B memory usage that are of paramount importance for the algorithms presented in this paper:

- All of memory can be represented as a DAG (directed acyclic graph) with a single entry point.
- Once created, a node is never changed (except for the purposes of maintaining reference counts). It may be copied, or recovered by the memory manager, but not altered.

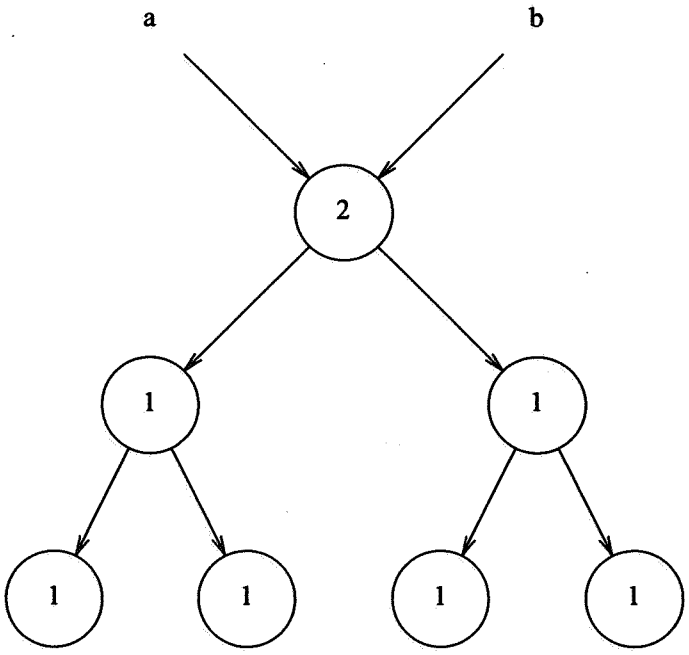

Figure 2: Two identifiers sharing the same values
(numbers denote reference counts)

INTERNAL REPRESENTATION
Internally, all nodes in memory can be represented as follows:

reference count	tag	... values ...

The reference count field is used by the memory manager to determine when storage for the node can be recovered. The tag field describes the nature of the object; that is, the correct interpretation for the value field. Various tags might identify an object as a number, a text, a data structure (such as a table), or an internal type such as a binary tree node.

Some object types, such as data structures or internal nodes, use the value field to point to other objects in memory. As portions of memory are moved to disk, these pointers must be updated to reflect the address on disk, rather than the address in memory. There are difficulties with this, not the least of which is that on many machines disk addresses can be considerably larger than memory addresses. A technique that will circumvent this problem, as well as solving several other difficulties we will subsequently encounter, is to use an auxiliary table called the *disk-mapping* table.

The disk-mapping table (Figure 4) is a statically allocated table of fixed sized nodes. The entries in the table are similar to other nodes in memory, in that they have a reference count and a tag field. However, only two types of tags can appear in these nodes; one indicating the node is in use and one indicating that it is free. There are two entries in the value field for the node; a memory pointer to a node and a disk address for the same node on disk.

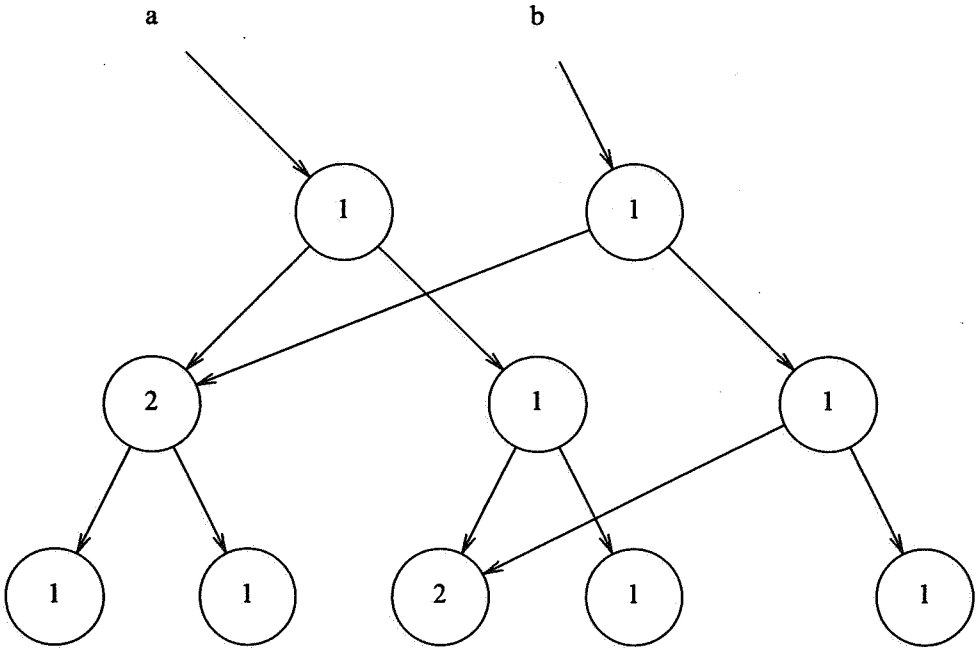

Figure 3: Identifiers after assignment of *b*

reference count	tag	disk mapping table memory address	disk address
0	free	–	–
3	used	2256	13427
1	used	–	14277

Figure 4: A portion of the disk-mapping table

The table is used as an associative mapping of both memory address and disk address. Using the memory address as a key, it is possible to search the table to find the associated disk address entry. Similarly, it is also possible to search the table using the disk address as the key, to determine if any entry corresponding to that address remains in the memory.

A node can be in memory only (in which case there is no entry in the disk-mapping table, and all pointers point directly to the node), both in memory and on disk (in which case the entry in the disk-mapping table maintains both addresses), or on disk only. In the latter case the memory field in the disk-mapping table is null, and only the disk address is used.

We add two bits to the tag field for every node. The first bit, the *copied* bit, is set by the cleaning person when a copy of the node has been moved to disk. The second bit, the *recently-used* bit, is turned off by the cleaning person but turned on by the B system. Recently used nodes are left in memory, with the presumption that they are likely to be used again soon and thus speedy access should be maintained. If the cleaning person encounters any node twice in succession without it being used by the B system, the pointer to the node will be changed to point to disk, rather than

memory.

When a node is copied to disk, initially pointers to the node remain pointing to the copy of the node that is still in memory. If the node is not used by the B system, these pointers are eventually changed to point to the disk-mapping table. As not all pointers can be found at one time, the original node is left in memory with the copied bit set to one. If in wandering through memory the cleaning person encounters a node with the copied bit on that has not been recently referenced, the pointer to the node is changed to point instead to the disk-mapping table entry (Figure 5). When all pointers to the original node have thus been modified (indicated by the fact that the reference count for the original node becomes zero), the storage for the node can be recovered by the memory manager. When all the nodes that themselves point to a node on disk have also been moved to disk (indicated by the reference count in the disk mapping table being reduced to zero), the entry in the disk mapping table can be reused.

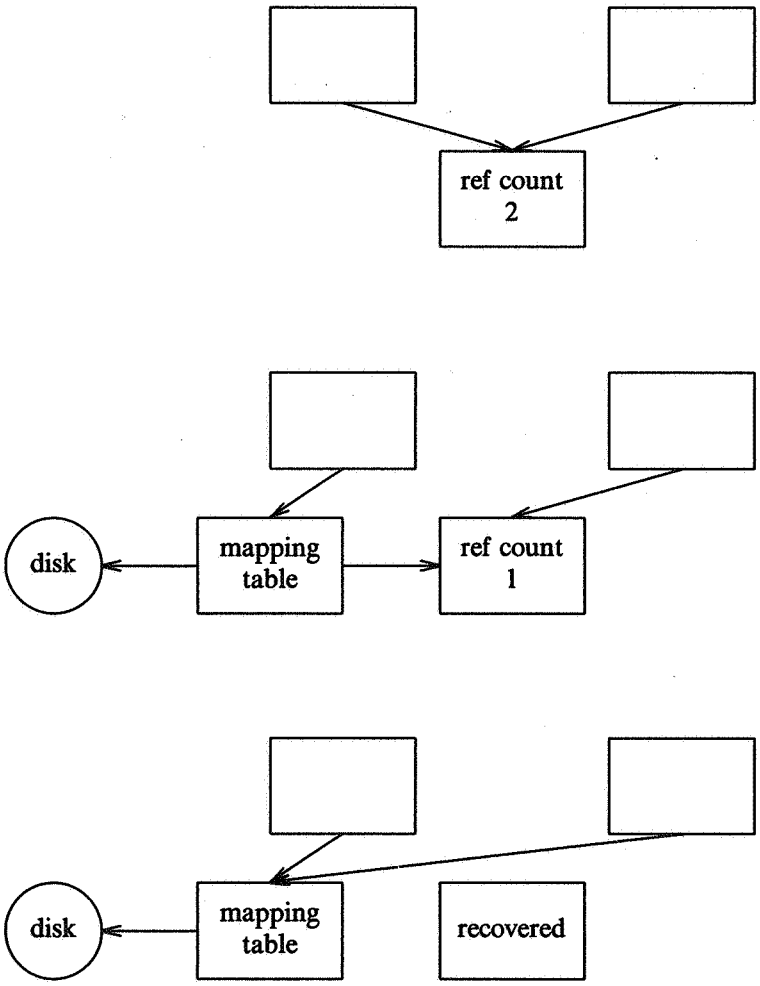

Figure 5: Sharing in memory becomes sharing on disk

Although externally entries in the disk-mapping table resemble all other nodes, they have several special characteristics. The most obvious is that they are not recovered and placed in the common pool of free storage by the memory manager when their reference count reaches zero[2]. Less obvious

2. A possible trick here is to initialize all the table entries with a one in the reference count field. Thus the reference count can never be reduced to zero, and the memory manager would never be invoked to free a mapping table entry. This would also el-

is the fact that the memory pointer in each entry can point to a node in memory without incrementing the reference count for that node. This is so that when all other references to the node have been eliminated the memory manager will automatically be alerted. If the memory manager is called upon to recover a node on which the copied bit is set, it searches for the associated entry in the disk-mapping table and sets the memory pointer for the entry to null before releasing the storage.

Finally, it can happen that an entry in the disk-mapping table can have a zero reference count and still not be removed. If the memory pointer for the entry is non-null, it means that there remain pointers in memory that have not been modified by the cleaning person to point to the disk-mapping table. These may subsequently be found and changed, thereby incrementing the reference count for the disk-mapping table entry. Thus as long as an entry in the disk-mapping table has a valid memory pointer field, it is not removed. When the memory pointer is eliminated and the reference count is reduced to zero (these actions can happen in either order), the tag on the entry is changed to reflect the fact that the entry is no longer being used[3].

Discussion of the Cleaning Person Algorithm

The algorithm we will describe can be viewed as a trio of interacting processes. The first process represents the B system, a more or less conventional interpretive system, which is allocating and using memory. Thus the B system sits on top of the memory manager, which controls the allocation of memory and tries to discover when memory is no longer being used. In parallel with these a third process, the *cleaning person*, attempts to move nodes from primary memory (RAM) to secondary storage (disk), thereby freeing nodes for reclamation by the memory manager. In contradiction to what we said earlier, the cleaning person is allowed one power that the B system is not; namely the cleaning person can change nodes in place. (Without this, the pointer semantics would insist that only a copy of the changed node be created, and no storage would ever be reclaimed, thereby defeating the purpose of the entire exercise.)

From time to time the B system creates a copy of the root node for the global environment and starts the cleaning person processing. Since the pointer semantics insure that this tree will not be subsequently altered, the cleaning person can run in parallel with the executing B system with no difficulty, and is free to move whatever nodes it can to disk. In the exceptional circumstances that the memory manager runs out of memory to allocate, all computation can be halted while the cleaning person attempts to move a sufficient amount of memory to disk to permit the memory manager to continue.

Because a complete image of memory as it existed at some point is moved to disk, the disk can also be used to recover from catastrophic events, such as a machine crash. In order for this to be feasible, it is only necessary that the pointer to the start of the memory DAG be known. Thus each time the cleaning person completes one entire sweep through memory, this pointer is stored at a fixed location on disk, such as the first word in the file.

Since actual parallelism may not be possible on all systems for which the algorithm is desired, the cleaning person process will be described as a loop. It is not terribly important that each iteration of the loop be executed *immediately* following the previous iteration. Thus at various points the B system can merely call the cleaning person process, which will perform a single iteration of the loop and return. A convenient place for this might be while the B system is waiting for input from the user terminal.

The cleaning person makes a postorder traversal through memory, moving all nodes to disk. Since back-pointers to parents can not be included in nodes without serious performance penalties (at any

iminate the anomaly introduced by sometimes not freeing an entry with a zero reference count, although it would introduce another curious situation where sometimes nodes with a reference count of 1 would be considered "unused."

3. Since an entry is unused if and only if the reference count is zero and the memory address field is null, one can actually avoid using two types of tags, merely using one tag to let the memory manager know the node cannot be recovered in the normal way.

point the number of back pointers required would be equal to the reference count of the node) it is necessary for the cleaning person to keep a stack indicating the parents of the node currently being considered. As a pointer to a node is placed on the stack the reference count for the node is naturally incremented. It is therefore not possible for the reference count on a node to be reduced to zero if any subtree rooted at that node is being examined by the cleaning person. This prevents the difficult circumstances where the memory manager would like to recover a node the cleaning person is currently processing. Since a node can have multiple children, the stack maintained by the cleaning person has a second field, indicating the number of children of the node that have already been examined.

The Cleaning Person Algorithm

As we have previously noted, the cleaning person algorithm consists of a succession of calls on the following routine. Each call progresses the cleaning person one step, and leaves it in a state ready for the next call. Initially the cleaning person stack consists of just the single entry pointing to the root of the memory tree.

```
cleaning person algorithm

examine the topmost entry on the stack

SELECT:
    some child has not yet been visited:
        increment the counter keeping track
        of the number of children visited
        visit next child

    all children have been visited:
        pop entry from stack
        IF copied bit is off:
            append copy of node to disk
            set copied bit in node
            make a new disk mapping table entry
```

Note that the pointer to the node in the parent is not changed at the time the node is moved to disk. This pointer will be changed the next time the cleaning person visits the node, unless it is referenced in the interim, in which case it will remain pointing to memory.

Moving a node to disk is slightly complicated by the fact that under no circumstances should a node on disk point back into memory. A consequence is that a node can only be moved to disk after all its children have been moved to disk. Notice that despite children being moved to disk, the pointers in the parents will not be changed until it is determined that they are not being used. Since the pointers in the node will still point into memory, whereas the pointers in the image on the disk should contain actual disk address, and since disk addresses may take more bits to represent than memory addresses, a temporary buffer is used to construct the image that will be copied to disk. On the other hand it makes no sense to maintain reference counts on disk, so this field need not be copied.

Subtrees that have already been moved to disk do not need to be visited. An exception to this rule are subtrees that have been recently used, and which therefore maintain their pointers in memory. Since child subtrees may not have been recently used, they are investigated. Thus the algorithm for visiting a child is as follows:

visit child algorithm

IF child is not an entry in disk mapping table:
    SELECT:
        copied bit on **AND** recently used:
            turn off recently used bit
            push child on the stack

        copied bit on **AND** not recently used:
            find associated disk mapping table entry
            modify the parent pointer[4]

        copied bit not on:
            push child on the stack

ANALYSIS

In order to argue why there can be no catastrophic pathological interaction between the B system and the cleaning person, or between the memory manager and the cleaning person, it is first necessary to give an abstract description of the behavior of these two protagonists.

As far as the cleaning person is concerned, there are only two actions an executing B system can take that are of any interest. These actions are accessing a node and modifying (or creating) a new node. We discuss each of these in turn.

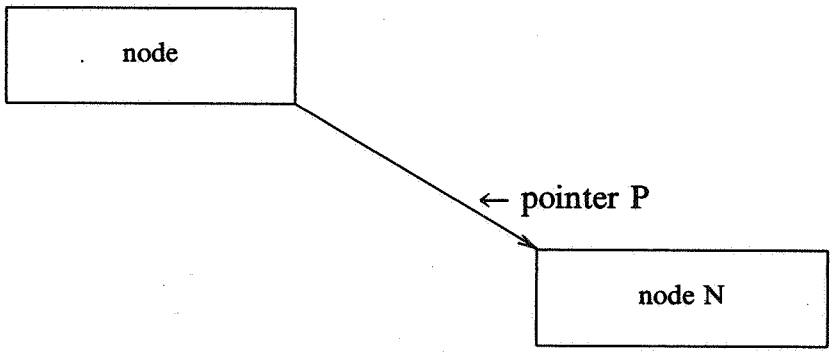

Figure 6: Interaction between the cleaning person and the B system (see text for explanation).

Suppose the B system desires to access a node N pointed to by a pointer P (Figure 6). Clearly any actions the cleaning person takes prior to the B system obtaining P are not relevant to the discussion, nor are any actions taken following the end of processing of N. Thus the only serious interaction must occur between the time the B system acquires a pointer and the time it uses the storage pointed to by that pointer. Consider the following scenario:

1) The B system acquires the pointer P, moving it into a local variable.
2) The cleaning person moves node N to disk, changing the node containing P. Assuming this is

4. The pointer to the parent node is at this point the topmost node on the stack.

the only reference to N, the memory manager reclaims the memory for N. Although the value for the pointer in the node containing P has been altered, the value in the local variable created by the B system has not been modified.

3) The B system tries to dereference P and use the node to which it points. The information at this location in memory, however, has been completely modified by the memory manager.

Notice that a problem only arises because the memory manager has recovered the storage, not because of any actions of the cleaning person, *per se*. Thus to avoid this problem it suffices to increment the reference count for node N immediately, in step 1. The actions of the memory manager will then be held up until after the processing of the node has been completed and the reference count once more decremented, following step 3. It is a relatively easy matter to insure that the B system adheres to this protocol.

Because of the pointer semantics used in B, it is not quite accurate to say that the B system is permitted to modify a node. Rather, a new node is created and the action is that of changing the pointer to point to this new node. Since this, however, involves changing the parent node that contained the pointer, a recursive argument percolates upwards until we find that most modifications require the regeneration of all nodes extending from the root of the memory tree down to the modified node[5] (as in Figures 2 and 3). Since the original tree being processed by the cleaning person remains unchanged, no problems can arise.

With regard to interaction between the cleaning person and the memory manager, we have already noted that when the memory manager tries to recover the storage used by a node it must examine the copied bit for that node. If the copied bit is on, it must then search the disk mapping table to find the associated entry, and nullify the memory address field for that entry. If the reference count for the disk mapping table entry is zero, this may also have the effect of freeing that location.

As we have noted, any entry pointed to by a value on the cleaning person's stack must have a positive reference count, and thus cannot be reclaimed by the memory manager. Even after all the children of a node have been visited, and the node is itself popped off the stack, its reference count still can not be reduced to zero since a pointer to the node exists in the parent, which is still on the stack. Thus a node cannot be reclaimed by the memory manager until the cleaning person is completely finished with it. Finally, the memory manager creating a new node will result in no interaction with the cleaning person, and thus is no problem.

Bringing a node back into memory

It can easily happen that a node, having been moved to disk by the cleaning person, must be brought back into memory in order to be used by the B system. In many respects, this is similar to an explicit modification of memory, such as results from an assignment. In both cases the pointer semantics insure that the original nodes are left unchanged. Note that in order for a node to be brought back into memory at all a disk mapping table entry must already exist for it. As the node is brought back, disk mapping table entries are created for all children of the node, and the node has its copied bit set. The latter action insures that if it is encountered once more by the cleaning person, it will not mistakenly make a duplicate copy of the node on disk.

How the Cleaning Person Algorithm is Used

Among the distinguishing features of the B system is the concept of the session record. Like the history mechanism for the Unix C-shell [3], the session record is a mechanism for recording the sequence of commands issued to the B system by the user. Of course, because of space limitations not all commands can be remembered, but only the most recent ones. The number of remembered commands, however, is sufficiently large (say 100), that this is not usually a problem.

5. In practice this is not as bad as it might seem. There is a trivial optimization that permits nodes to be modified in place if all reference counts from the root of memory to the node have value 1. As this can only happen if the cleaning person is not processing any portion of the tree in question, it has no bearing on the current discussion.

Among the more interesting features of the session record is the fact that it can be edited by the user, and restarted at any point. For example suppose the last two commands entered by the user were simple assignments, as shown in Figure 7 (the characters >> represent the B system prompt). At the most recent point in the session record, therefore, the value of the identifier *a* is 3 and that of the identifier *b* is 8. The user could at this point move back in the session record and change the first command to assign 7 to *a*, rather than 3. In so doing, the value of *b* would *automatically* be changed to 12.

```
>> ...
>> PUT 3 IN a
>> PUT a + 5 IN b
>>
```

Figure 7: A portion of the session record

As part of the implementation of this feature, the global environment (the value of all variables known at the command level) is saved following each command. As it would be rather costly to save all these copies of identifiers in memory, the cleaning person is called in to move as much as possible to disk. As we have noted, the pointer semantics of B permit these copy actions to take place in parallel with the execution of the following command.

The cleaning person will also be started working on all of memory (global and local variables combined) if memory should become too full, say more than one third full. Finally, all computation will halt and the cleaning person will be invoked continually if the memory manager should find itself unable to honor a request for more storage.

It can happen, although it should be rare, that the cleaning person will fall too far behind. In this case, as in the case where too much of memory is being used, the B system simply halts and waits for the cleaning person to catch up.

Reducing Disk Access

While the recently-used bit attached to each node is intended to reduce as much as possible the unnecessary retrieval back to memory of nodes that have been moved to disk, nevertheless, some disk I/O to return nodes is inevitable. However a simple virtual memory scheme can be used to reduce the number of disk access by increasing the amount of memory retrieved in each access.

In this scheme, an array of buffers is placed between the routines performing logical input and output and those performing physical I/O. These buffers are of some size convenient for the device, for example a multiple of the machine page size. Successive write commands merely append to the end of a buffer. Only when a write would exceed the size of the buffer is the buffer actually written out to disk. Similarly, when a read is requested a test is first performed to see if the node is in memory in some buffer. If so, the value is merely copied from memory. Otherwise, an entire buffer is retrieved from disk.

Because of the way the cleaning person travels through memory visiting nodes, it is likely that if a node is found in one buffer, at least some, if not all, of the children for the node will be found in the same buffer. Conversely, when a node is returned to memory there is a nontrivial likelihood that at least some of the children for that node will also be brought back into memory. Therefore this simple buffering scheme, while not complicating either the cleaning person, the memory manager, or the B system, can reduce the need for repeated disk access to retrieve small nodes.

Implementation of the Disk Mapping Table

Since the Disk Mapping Table is consulted rather often, one would like access to it to be as fast as possible. A simple, but effective, approach is to use two hash tables, each of which points to a position in the fixed size disk mapping table. The first hash table is keyed on memory addresses, while the second uses the disk address for the key. Access can be accomplished very quickly by either function. Once an entry has been found, both the memory and the disk addresses are known, thus entries in both hash tables can be located. This is necessary for removal of entries once they are no longer needed, for example.

The Growth of the Virtual Memory File

As the cleaning person algorithm has been described up to this point, it would appear that the Virtual Memory File would be permitted to grow without bounds as execution continued. In a practical situation this is, of course, not realistic. However a simple scheme which depends upon the particular way in which the cleaning person algorithm is used can prevent this file from growing unnecessarily.

As was noted in a previous section, the information stored in memory and on disk consists of trees corresponding to the state of memory for the previous N steps, where N is a number around 100. The user can, if desired, back up execution to any point in the previous N instructions, in which case the stored image of memory becomes the current memory.

One possible scheme to remove unusable storage from the Virtual Memory file would be to adapt a conventional memory allocation algorithm, such as reference counting or garbage collection, for use on the disk. This would, however, impose unacceptable performance penalties because of the necessity for repeated disk access. A simple scheme splits the Virtual Memory file into three separate files. A counter is maintained, and every N/2 times the cleaning person is called upon to write out a new tree the following actions are taken:

- The oldest of these three files is discarded. Since the memory trees it contains are now more than N steps old, they are of no further interest.
- A special flag is set for the cleaning person. As the cleaning person traverses memory the first time after starting a new file, all nodes, whether in memory or on disk, are copied to the new file. After traversing the memory tree this flag is reset, and thereafter only nodes in memory will be considered by the cleaning person.

The latter action insures that if a root of a memory DAG is moved to disk, then all children from that tree will also be found on the same file. Thus it is only necessary to remember on which file the root nodes appear. This change requires only small modifications to the algorithm we have developed to this point, and with small cost eliminates the unbounded growth of the Virtual Memory file.

Acknowledgments

I wish to thank Lambert Meertens for suggesting the problem and Martha Visser for first inspiring the solution. The cleaning person algorithm was developed and refined over the course of several meetings of the B group at the CWI, and their criticisms and help are gratefully acknowledged. Lambert, in particular, was an incessant prod in proposing ways in which the algorithm could be simplified, complicated, or improved (often all at the same time!). Other helpful suggestions were made by Steven Pemberton and Guido van Rossum.

References

1. L. Geurts, An Overview of the B Programming Language, or B without Tears, *SIGPLAN Notices 17*, 12 (Dec. 1982), .
2. L. Geurts, L. Meertens and S. Pemberton, *The B Programmer's Handbook, Centrum voor Wiskunde en Informatica, Amsterdam, 1985.*
3. W. Joy, An Introduction to the C shell, in *Unix Programmer's Manual*, vol. 2C, August 1983.
4. T. Krijnen and L. G. L. T. Meertens, Making B-Trees Work for B, IW 219/83, Mathematisch Centrum, Amsterdam, 1983.

5. L. MEERTENS, *Draft Proposal for the B Programming Language*, Mathematisch Centrum, Amsterdam, 1981.
6. L. MEERTENS and S. PEMBERTON, Description of B, *SIGPLAN Notices 20*, 2 (Feb. 1985), 58-76.
7. L. MEERTENS and S. PEMBERTON, An Implementation of the B Programming Language, Note CS-N8406, Centrum voor Wiskunde en Informatica, Amsterdam, 1984.