AD-A196 236

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4147

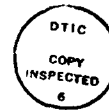
Title: The SMITE Object Oriented Backing Store

Author: S R Wiseman

Date: March 1988

SUMMARY

The SMITE computer system is to be provided with a write once backing store that allows objects to be stored permanently. Capabilities are used for addressing objects and on the fly garbage collection is used to recover inaccessible objects. This paper describes the proposed implementation of the backing store and its garbage collector.




Accesion For
NTIS CRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution /
Availability Codes
Dist | Avail and/or Special
A-1

Copyright
©
Controller HMSO London
1988

## 1. Introduction

The SMITE computer system is being developed to provide a base for high assurance multi level computer security applications [Wiseman86]. The computer provides a vehicle for executing the Ten15 programming environment [Core&Foster86], using the Flex capability architecture [Foster et.al.82] to give highly assured protection.

The Flex computer has demonstrated the effectiveness of a write once, object oriented backing store [Currie et.al. 83] in a software engineering environment. The SMITE backing store follows the same approach, but has two additional features to accomodate security requirements. One extends the backing store to allow instances of user defined, protected abstract types to be stored and the other provides a quota mechanism and on the fly garbage collection.

The main advantage of a write once backing store, in software engineering terms, is that it can be kept consistent. An additional advantage for security applications, is that objects in the backing store can be guaranteed to be unalterable and hence do not provide a communication path.

## 2. Backing Store Organisation

The SMITE backing store allows objects to be stored permanently. These objects may contain scalar data, such as integers, and capabilities for other objects in the backing store. Various types of object can be stored, including procedures, objects representing types and objects which are instances of user defined abstract types.

The backing store is organised on a write once basis. To store data in the backing store, only the data is provided, because unlike conventional filing systems the user does not state where the data is to be stored. The data is placed on disc at some free place and a capability for the new backing store object is returned. Data is retrieved from the backing store using the capability for the data. Unless the appropriate capability can be obtained, the data cannot be retrieved.

Backing store capabilities may reside in main memory, however the representation is different to that on disc. When a backing store capability is brought into main store, it is represented as a main store capability for an instance of an abstract data type (a backing store capability), which holds details such as the object's location. However, main store capabilities cannot be placed in the backing store. This is because the lifetime of main store objects is much shorter than those in the backing store. When the computer is turned off, or crashes, main store objects are lost but backing store objects persist. Thus main store capabilities in the backing store would become inconsistent.

Dictionaries (directories) and other modifiable structures cannot be constructed using a write once backing store. The backing store therefore provides special objects, called references, which can be altered. References are small objects which can contain a capability for some backing store object (though they can alternatively contain a scalar value). They can be atomically updated, that is the update operation will appear to either successfully complete or fail without changing anything, even in the face of errors such as power failure. The update operation requires that the existing value of the reference be supplied as a parameter. This permits the implementation of dictionaries that allow multiple updates to proceed in parallel, using a simple re-try mechanism to resolve conflicts.

There is one special reference, called the *root reference*, which is stored in a known place. When a backing store is first brought on line, a capability for the root is manufactured by the backing store software. That is, an instance of the backing store capability abstract type is produced, which contains the address of the backing store root on disc. The root refers to the accessible structure held in the backing store. Any objects which cannot be reached by following capabilities from the root can never be accessed and such garbage is recovered by the garbage collector. Note that the backing store has no explicit delete operation. In practice, the backing store root would refer to something like a dictionary of dictionaries, though as far as the backing store implementation is concerned it is used in the same way as any other reference.

To allow simple quota management and to limit the effects of one user on another, the backing store is divided into logical areas. Each backing store object is allocated in one of these areas. Each area can be limited in the total amount of physical store allocated to it, including both accessible and inaccessible objects, free space and space lost due to fragmentation. Garbage collection is performed on the fly, while the disc is on line, though each area is garbage collected independently. Garbage structures which span area boundaries are dealt with using the algorithm of [Wiseman85].

Backing store capabilities exist in two forms, *firm* and *shaky*. Firm capabilities are the usual sort, shaky capabilities behave in the same way except that they do not prevent an object from becoming garbage. If all the accessible capabilities for an object are shaky, the object is deemed to be garbage and all the capabilities for it are turned into a nil capability (which appears as the scalar zero when loaded into main store). Operations are provided to convert firm capabilities to shaky, and vice versa. Shaky capabilities are sometimes used by resource manager software for keeping track of an object while it is being used, but without preventing it from being discarded.

The backing store therefore provides permanent storage for structured objects, uses capability addressing and operates in a write once fashion. Unlike conventional systems it does not provide a directory structure, since any required facilities can be built using SMITE's type abstraction mechanisms.

## 3. Backing Store Interface

The interface to the backing store is given by just a few simple procedures. There is one procedure for storing objects, which returns a new backing store capability for the newly created backing store object, and one for retrieving stored objects. Other procedures convert between firm and shaky backing store capabilities, create new references and logical areas and perform garbage collection. Facilities for naming objects are built on top of these mechanisms and are not considered as part of the backing store interface.

Each procedure for creating new objects in the backing store has a capability for an area object bound to it (procedures are first class data values [Currie82]). The created object is assigned to that area for accounting and garbage collection purposes.

Before an object is loaded from the backing store into main memory, a check is made to see whether it is already present in the main store. This optimisation is possible because objects brought in from backing store are unalterable. The check involves searching a hash table which contains a list of the backing store capability and corresponding main store capability for all backing store objects that are currently in main store. The search is performed by a single SMITE instruction.

**store** : mainstore_cap → cap

**load** : cap → mainstore_cap

**firm_cap** : cap → cap
**shake_cap** : cap → cap

**make_reference** : word → cap
**update_ref** : cap × word × word → ()

**make_area** : () → cap
**update_area** : cap × ..... → ()

**garbage_collect** : area → ()

**size_of** : cap -> int
**type_of** : cap -> int
**alterable** : cap -> bool

**proclaim_unalterable** : cap -> cap

Fig 1. File Store Interface Procedures

The procedure **store** takes as a parameter a capability for some data in main store. The data must not contain any main store capabilities, otherwise the procedure will produce an exception. Data is retrieved from the backing store using the procedure **load**. This takes a backing store capability for any object and returns a main store capability for a copy of the data in main memory.

The type of main store object created by **load** depends on the type of main store object used to supply the data to the call of **store** that created the backing store object. The types are preserved, except that the result is always unalterable (read only). For example, a backing store object created from a data block will yield a read only data block when loaded and abstract type objects (keyed blocks) yield read only abstract types (read only keyed blocks).

An alternative scheme would be to give a copy of the object stored in the backing store. This may be more meaningful in respect of persistent variables. However, this does not affect the backing store organisation, which is the subject of the paper. Similarly, the interface could be extended to allow data to be gathered together from a variety of sources, for example arrays or structures. However, this could only be provided in a way which does not admit denial of service attacks.

Applying load to a backing store reference gives the current value stored in the reference, which is either a backing store capability or a 32' scalar value. Applying load to an area object gives a read only copy of its state information, such as quota details.

Backing store capabilities may be converted between firm and shaky capabilities using **firm_cap** and **shake_cap**. These simply copy the backing store capability and return a firm or shaky version as appropriate. If the object referred to by a shaky capability has become garbage, the firm operation returns the scalar zero.

New backing store references can be created using **make_ref**. The reference is initialised to contain the given scalar value or backing store capability. An existing reference can be updated, with a scalar word or backing store capability, using **update_ref**. This takes, as a parameter, the current contents of the reference. A check is made to ensure this is given correctly. If the check is passed, the new value is placed in the reference. Special precautions, described in section 6, are taken to ensure that the check and update is performed atomically and without danger of irreversibly corrupting the disc in the event of a hardware error such as power failure.

New area objects are created using **make_area**. The new area is initialised with some free space and is then ready for controlling the allocation of new objects. The quota information inside an area can be altered using **update_area**. Garbage collection of an area is performed, with the backing store on line, using **garbage_collect**. Access to an area object must of course be carefully controlled, using the capability mechanisms, to ensure that these operations are not misused.

Operations are provided which give information about backing store capabilities. **type_of** gives the type of main store object which would be created if the capability were loaded and **size_of** gives its size in bytes (in the same way as the block size instruction does for main store objects).

The function **alterable** can be used to determine whether the structured object which is accessible from a capability, can be altered in any way. That is, an object is alterable if it contains a capability for an alterable object, a reference or a shaky capability. This mechanism is used to ascertain whether a backing store capability may be shared without introducing covert communications channels.

The function **proclaim_unalterable** is highly privileged as it changes alterable capabilities to unalterable ones. This is necessary to allow systems to provide objects which, although they are alterable, protect themselves sufficiently to be treated as unalterable. For example, a bulletin board which can only be updated by cleared personnel using a trusted path [Wiseman88], can be freely shared.

## 4. Implementation

This section describes the implementation of the write once, object oriented backing store and gives details of the data structures held on disc. The backing store subsystem is implemented in software and protects itself by using the usual SMITE mechanisms of abstract data types and procedures [Wiseman88].

### 4.1 Disc Fragments

The disc is partitioned into fragments, which are a number of consecutive disc sectors (a sector is probably 512 bytes). Each fragment is either unallocated, allocated to a logical area for storage allocation purposes, a representation of a logical area or defective. The format of a fragment is shown in figure 4.1, except for defective ones which are simply ignored by the system.

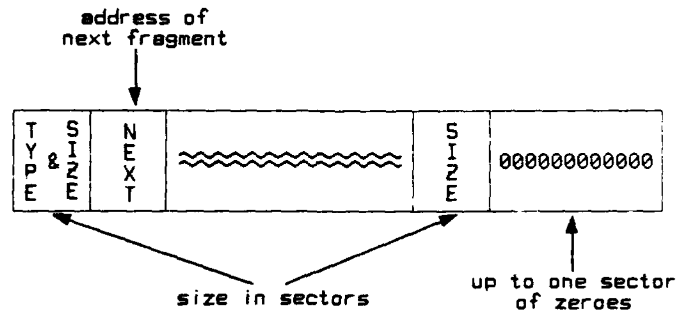



Figure 4.1: Disc Fragment Format

The first word of a fragment gives its type and total size in sectors. The type field gives some redundancy by distinguishing between unallocated, allocated and area fragments. The second word is the link field for threading the fragment onto a list. It contains the absolute sector address of the next fragment, or zero. In the last sector of the fragment, the size is repeated and the remainder is zeroed. Therefore by searching the last sector backwards it is possible determine the fragment's size and hence its start address. The padding zeroes are necessary because fragments may be split up during garbage collection and the remaining space may not be large enough for a garbage object.

The disc's header contains the list heads for a list of unallocated fragments and a list of fragments which represent logical areas, as well as the disc root. It also contains a map of the defective fragments, because by their very nature these cannot be linked together, to facilitate recovery if the disc is ever corrupted. The header also contains space for recording the progress of atomic transactions.

## 4.2 Allocated Fragments

Each logical area has a list of fragments which it uses to allocate objects. Objects are allocated one after the other in the list, so that the oldest object appears at the start of the first fragment and the youngest towards the end of the list. Beyond the youngest object is space available for allocating further objects. The logical area records the next free place, as a fragment address and an offset, as well as the list header.

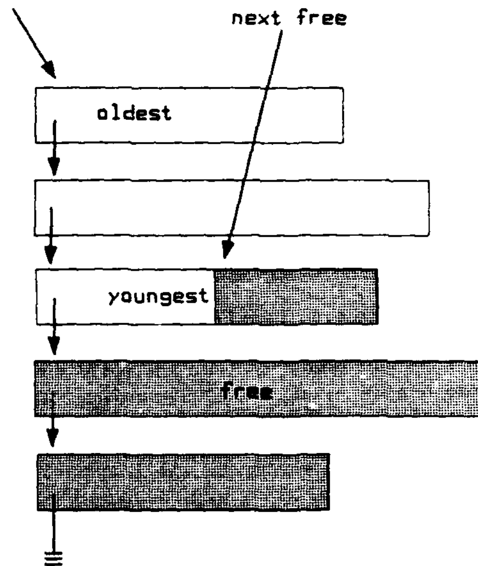



Fig 4.2: Objects are allocated one after the other.

Note that the next free location is not updated each time an object is allocated. This is because the object will become garbage when the computer is turned off, unless a capability for it is stored somewhere which is preserved across system start ups. Therefore next free is only updated when a capability is placed into a reference or is stored in a different area (this avoids having to update the next free location of all logical areas whenever a reference is updated).

It is important to note that, because new objects are allocated in the next free space along the storage, the objects allocated in a logical area are stored in order of creation.

When the garbage collector recovers space occupied by an object, it is linked onto the end of the fragment list where it can be used for allocating new objects. Note that space occupied by garbage can only be recovered if an entire disc sector becomes garbage. While any part of a disc sector is occupied by an accessible object, the remaining free space cannot be recovered.

4.2 Backing Store Objects & Capabilities

Objects are stored, as shown in figure 3, starting with their type and the size (in words) of main store block required to hold them. The data in a flattened form follows this header. Finally a word giving the total size of the object on disc is stored. This allows the garbage collector to determine where the object preceeding a particular object starts, allowing it to scan from younger to older objects.

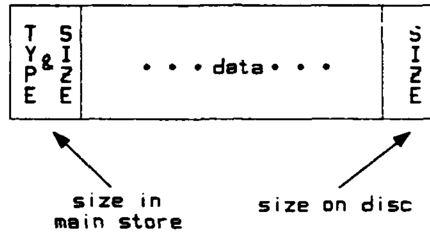



Figure 4.3. Data on Disc

Each word of the data is either a scalar value or a backing store capability. The words are tagged to distinguish between these. The tags for each set of thirty one (main store) words are collected together and stored before the representation of the words, as shown in figure 4. If the word is a scalar it is simply stored as four bytes. If the word is a capability it occupies eight bytes, because addresses on disc are larger than in main store. If the object is not a multiple of thirty one words, the last set will be incomplete, though thirty one tags will still be stored. The remaining tag bit in each tag word is used to indicate that the data is split across fragment boundaries.

It would be possible to omit the tags if the object is of a type that never contains capabilities. However, if they are always stored, the case when an object crosses fragment boundaries, is simpler.

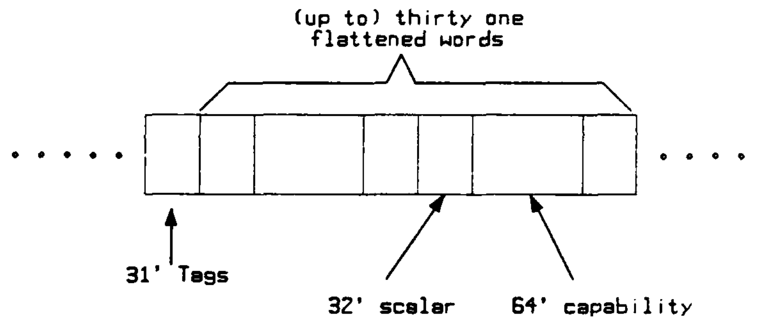



Figure 4.4. Flattened Tagged Data

Capabilities stored in the backing store occupy eight bytes. This comprises a thirty nine bit word address (allowing for a 2Tbyte disc), a twenty bit size field and a four bit type. The size field gives the size, in words, of the object's representation in main memory. It is a copy of the value found at the start of the object and is used to allow the size to be determined without accessing disc. Similarly for the type field. Another bit indicates whether the capability is stored in a logical area different to that which the object it refers to is allocated.

Figure 4.5: Format of a Capability in the Backing Store

Objects are stored contiguously along the list of fragments (Figure 4.6). However, an object which would cross a fragment boundary is split up into smaller pieces. A tag word whose most significant bit is set is used to indicate that the object is continued in the next fragment. When this is first written to disc, it is necessary to write the fragment's size into the next word. This allows the start of the fragment to be determined, and hence the address of the next fragment, when the object is read.

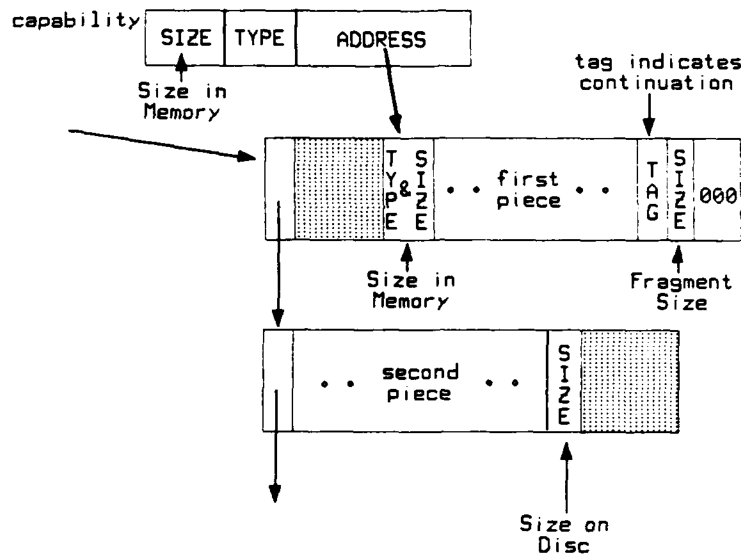



Figure 4.6: Object Crossing Fragment Boundary

## 4.3 Inter Area Capabilities

Capabilities for objects belonging to other areas contain an indirect address which refers to a small object that contains the true address. The indirection allows an area to be garbage collected and compacted independently of other areas. Thus each logical area has a list of indirection objects for incoming capabilities.

The indirection objects contain a capability for the object which is referenced, the number of the logical area it resides in, some garbage collection flags and a pointer to the next indirection table entry. This pointer comprises the address of the fragment containing the next entry and an offset. This allows the entry to be recovered more easily when it is no longer required.

## 4.4 Backing Store References

A reference is a variable that can contain either a backing store capability or a scalar value. Each logical area has a list of the references allocated to it. The reference object therefore contains a capability or scalar value, some garbage collection flags and a pointer to the next reference.

## 4.5 Area Objects

An area object contains the physical address of an area fragment. If the garbage collector can recover an area object, the fragment is recovered as well once it is empty.

### 4.6 Garbage Objects

It is sometimes not possible to recover the space occupied by an inaccessible object. This fragmentation arises because only entirely empty disc sectors can be reused. Garbage objects are created to cover the inaccessible but unrecoverable space.

## 5. Garbage Collection

Garbage collection is the mechanism used to recover the disc storage occupied by objects that are no longer accessible. The backing store offers no explicit delete operation, instead an object is kept only as long as it is possible to access it.

### 5.1 The Garbage Collector

The garbage collection of an area is performed on the fly, while the area is still being accessed, independently of all other areas. The garbage collection of each area contributes to the garbage collection of the overall backing store using the algorithm of [Wiseman85].

An area is garbage collected by first identifying which objects are directly accessible, the so called roots of the heap. These are objects which are referred to by capabilities in main store, found by searching the hash table, and by the incoming capability list. The set of objects which have yet to be scanned is maintained, and is initialised to the set of root objects. The garbage collector visits each of the objects in the area in turn, starting with the newest and progressing to the oldest. Another set is created, which records those objects that are garbage. Initially this is the set of all objects. Gradually, as the scan finds objects to be accessible, this set is reduced. The scan ignores any references and incoming indirection objects that it encounters.

If the object visited is in the set of objects that need scanning, it is removed from the set, scanned for capabilities and removed from the set of garbage objects. Otherwise the search goes on to visit the previous object. For any capabilities found by the scan, the objects they refer to are added to the set for scanning. These will always be older objects, because of the write once nature of the backing store, and so will eventually be visited and themselves scanned. If the set of objects to be scanned becomes empty (this will certainly happen if the oldest object is reached) the scan finishes, and the garbage set indicates which objects can be recovered. If any capabilities for objects in other areas are found by the scan, the incoming capability list they refer to is marked to indicate the object is accessible.

If insufficient workspace is available to maintain the set of objects to be scanned, garbage collection must halt. However, some garbage in the backing store may still be recovered, as long as all those objects not yet visited are assumed to be accessible. In fact it may be advantageous to stop the garbage collection early and recover whatever garbage has been discovered so far. This is because older objects tend to be much longer lived than those newly created [Baker78]. Hence by garbage collecting the younger part of the store more often, more garbage should be recovered for the amount of scanning performed.

Note that the set of garbage objects need not be constructed. An object is known to be garbage as soon as the scan reaches it, if it is not in the set of objects that need scanning. In this case it is possible to immediately recover the space it occupies. It is, however, more profitable to collect together a series of adjacent garbage objects and recover the space in one larger lump.

Store is recovered by locating contiguous sectors of disc storage which contain only garbage objects. These contiguous sectors are removed from the fragments containing them and are added to the free end of the fragment chain as new fragments. Removing parts of a fragment involves updating several parts of the disc, which must be performed atomically, see section 6. Note that garbage cannot be recovered if part of the same disc sector contains accessible data. This is the cause of potentially serious fragmentation, overcome by using the compaction algorithm described in secton 5.3. Also, there must be space around the accessible objects to fit fragment headers when necessary.

## 5.2 Shaky Capabilities

Shaky capabilities are those which do not prevent an object from becoming garbage. The garbage collector handles shaky capabilities by noting whether an object has been found to be wanted by shaky capabilities alone. When the garbage collector reaches an object which is only referred to by shaky capabilities, the object can be declared garbage, because no firm capabilities will be found in older objects. However, all the shaky capabilities must be changed to nil (the scalar zero). This is achieved by placing a tombstone, using an atomic operation, on the site of the object [Lomet75].

Tombstones are recovered by the next cycle of garbage collection. Whenever the scan finds a shaky capability, a check is made to ensure the object has not been replaced by a tombstone. If it has the capability is replaced, in an atomic action, by nil (a special capability value which turns to the scalar zero when it is loaded). When the scan meets a tombstone, all shaky capabilities that refer to it will already have been found and changed to zero. Therefore the object can be recovered.

## 5.3 Compaction

A logical area of the backing store may be compacted provided there is sufficient free storage on the disc available as a temporary workspace. This recovers unused space which is unavailable because it exists on the same sector as accessible storage. Compaction is achieved by copying the area's data objects into the free space. It can be added as a final phase of garbage collection, in which case only accessible objects are copied, or it may be performed at any time by copying all objects. Note that references and indirection objects cannot be compacted in this way because they are held on linked lists. An alternative scheme for them is described later.

The compactor visits each object in turn, progressing from the oldest to the youngest. If the object is wanted it is copied into the free space. A table is maintained, using workspace in main store, of the new address of each object copied. This is used to update any capabilities that are copied. No capability can be encountered which refers to a younger object, and hence has not been copied to its new location, because of the write once nature of the backing store.

New objects, created during the compaction, are stored in the old area. When all the objects have been copied, the root capabilities can be updated to refer to the new versions of the objects. Note that this need not be one atomic action because the objects cannot be altered so both the old and new versions are equally valid. Those backing store capabilities held in main store must also be updated, though this need not be an atomic action as long as all reads deliver capabilities to the new versions. Once all capabilities refer to the new versions of the objects, the old space can be recovered, except for space occupied by references and indirecton objects.

If, during compaction, the relocation table overflows, the least recently used entries can be discarded. If a discarded entry is required later, it can be recomputed by scanning both the old and new copies of the objects together. If insufficient free store is available to copy the entire area, it is possible to compact just the youngest part of it. This is because no capabilities in the older part refer to objects in the younger part.

Compaction of references and indirection objects may be performed by creating new objects which are a copy of the original, and then placing a temporary indirection in the original object. Whenever such an indirection is found, the capability can be updated to refer to the new object directly. Eventually the garbage collector will ensure that all capabilities have been found and updated and the old objects can be recovered.

5.4 Global Garbage Collection

Garbage collection of each area is performed independently as described in section 5.1. The algorithm described in [Wiseman85] is used to combine the effects of these garbage collections to effect a garbage collection of the entire disc. This is necessary to recover objects which were once referenced by other areas but which are now inaccessible. A brief overview of the algorithm will now be given.

The indirection entries, which are used to implement inter-area capabilities, contain some garbage collection flags. These indicate the state of the reference during a garbage collection scan. The possible states are **not found**, **shaky found** (which is used to cope with shaky capabilities), **found** and **scanned**.

Initially all capabilities are marked as **not found**. The roots, which are those capabilities in memory and in the disc root, are then marked **found**. The scan, which is effected by the actions of the local garbage collectors, visits each **found** capability and finds any inter-area capabilities which are accessible from it, before setting the capability's mark to **scanned**. Any inter-area capabilities which are found to be accessible, and are marked **not found**, are changed to **found**.

Eventually no capabilities will be marked **found**, at which point the scan has finished. Then the indirection entries for any capability marked **not found** is recovered, and those that remain have their marks changed to **not found** ready for another scan.

The local garbage collections perform a part of the global scan by making two passes across the objects in their area. The first pass scans those objects which are reachable from capabilities in main store and incoming capabilities which are marked as **found**. The second pass finds those reachable from other incoming capabilities. Only during the first pass does any outgoing capability which is marked as **not found** become marked **found**.

Performing the local garbage collections in two passes saves scanning effort, but at the expense of storing the set of garbage objects. If insufficient storage is available, it will be necessary to perform additional scanning for global garbage collection.

## 6. Atomic Actions

Any alteration to the disc must be performed in a way which ensures data is never lost, even in the face of power failures which may cause rubbish to be written. In particular, updating references and manipulating the disc fragment lists must be atomic actions. They must either succeed completely or fail without changing anything.

When data is written to a fragment, it will not usually need to be as an atomic action. This is because the new object cannot be referred to permanently until after it has been created. However, if it occupies the first or last sectors of a fragment, it must be updated atomically to preserve the fragment list information that is stored there. Also if the sector contains an object which is referred to by a reference or by another area it must be updated atomically. This ensures that a permanently accessible object is never overwritten with bad data.

The atomic update of a single sector is achieved by first writing the address of the sector to be updated, then the data to be written and then the address again to three sectors in a special place on disc. The sector is then updated with the new data. Finally the third sector is cleared. When the disc is brought on line, a check is made to see if an atomic update failed as the disc was last taken off line. If the two addresses in the special place are the same, an update had failed and is retried. Note that this scheme ensures that no data is lost even if the wrong data, rather than simply bad data, is written to the special place due to power failure.

More complicated atomic updates are required to manipulate the fragment lists. Here up to three sectors need to be altered atomically, but a simple extension to the atomic update mechanism suffices.

## 7. Conclusions

The object oriented, write once backing store for the SMITE computer has been described. An outline of the implementation, which includes on the fly garbage collection and compaction, has been given. The write once nature of the filestore greatly simplifies the garbage collection and compaction algorithms.

The existing FLEX implementation, which runs on re-microprogrammed ICL PERQs, is a working demonstration of the effectiveness of the write once structured backing store. However it performs garbage collection offline, which is inappropriate for the applications proposed for SMITE, though is quite sufficient for a personal workstation.

Thanks go to Michael Foster, Ian Currie and Peter Edwards for the original ideas behind the backing store.

## References

H.G.Baker
List Processing in Real Time on a Serial Computer
Comms of the ACM
Vol 21, Num 4, April 1978, pp280..294

P.W.Core & J.M.Foster
Ten15: An Overview
RSRE Memo 3977
September 1986

I.F.Currie
In Praise of Procedures
RSRE Memo 3499
July 1982

I.F.Currie, J.M.Foster & P.W.Edwards
Kernel and System Procedures in Flex
RSRE Memo 3626
August 1983

J.M.Foster, I.F.Currie & P.W.Edwards
Flex: A Working Computer with an Architecture Based on Procedure Values
RSRE Memo 3500
July 1982

S.R.Wiseman
A Garbage Collector for a Large Distributed Address Space
RSRE Report 85009
June 1985

S.R.Wiseman
A Secure Capability Computer System
IEEE Symposium on Security and Privacy
Oakland, California
April 1986

S.R.Wiseman
Protection & Security Mechanisms in the SMITE Capability Computer
RSRE Memo 4117
January 1988

DOCUMENT CONTROL SHEET

Overall security classification of sheet ...... UNCLASSIFIED

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S) )

| 1. DRIC Reference (if known) | 2. Originator's Reference<br>MEMO 4147 | 3. Agency Reference | 4. Report Security Classification<br>U/C |
|---|---|---|---|
| 5. Originator's Code (if known)<br>778400 | 6. Originator (Corporate Author) Name and Location<br>Royal Signals & Radar Establishment<br>St Andrews Road, Great Malvern, Worcs WR14 3PS | | |
| 5a. Sponsoring Agency's Code (if known) | 6a. Sponsoring Agency (Contract Authority) Name and Location | | |

7. Title

SMITE object oriented backing store

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

| 8. Author 1 Surname, initials | 9(a) Author 2 | 9(b) Authors 3,4... | 10. Date | pp. ref. |
|---|---|---|---|---|
| WISEMAN S R | | | 1988.3 | 13 |

| 11. Contract Number | 12. Period | 13. Project | 14. Other Reference |
|---|---|---|---|
| | | | |

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The SMITE computer system is to be provided with a write once backing store that allows objects to be stored permanently. Capabilities are used for addressing objects and on the fly garbage collection is used to recover inaccessible objects. This paper describes the proposed implementation of the backing store and its garbage collector.

S80/48