Addressing in a Persistent Environment

P. A. Buhr
University of Waterloo

C. R. Zarnke
Waterloo Microsystems Inc.

ABSTRACT

An addressing scheme is proposed, called a structured addressing scheme, that will provide a consistent addressing structure for a persistent environment. A structured address is the internal analogue of a qualified name in a programming language (e.g. x.y.z) where each component of the address identifies a special kind of data item, called a memory, whose basic purpose is to hold/store possibly many objects. Memories are organized in a hierarchical fashion. Some memories correspond to physical storage media, but others exist for organizational reasons. Each component of a structured address is not defined by the hardware, but by the memory in which the object is contained. Hence, a component can be encoded by a memory (so it is not a direct displacement) to reduce the length of an address. The component can point indirectly to the next component or object in the memory, making it a handle. This indirection permits the object to be moved around within its memory for storage management reasons and yet still maintain a fixed address. Finally, the structured address has the advantage that it accommodates new memories easily.

1. INTRODUCTION

A **persistent environment** is a programming system in which users can create data items of any type that outlast the execution of the program that created them, and hence, persist until the user explicitly deletes them or an entity containing them. Persistent data has always existed in programming systems, normally in the form of files and databases. What characterizes a persistent environment is the way in which this persistent data is made accessible in a programming language and its execution-time environment. A persistent environment simplifies some of the difficult problems in existing systems and reshapes others so that they can be solved in a more consistent and coherent form. Work done in traditional systems is recast into a form that combines together ideas from programming languages, operating systems and databases to provide a powerful software development environment, with increased software reliability, more efficient program execution and more productive programming time. Several different approaches have been presented to achieve this goal [1, 2, 4, 10, 11, 12].

What is common across all these designs is that the processor(s) on which they execute deals only with addresses to one type of memory (usually volatile RAM), which has the basic property that it be able to respond at approximately the same speed as the processor (called **primary storage**).

In John Rosenburg and David Koch, editors, *Persistent Object Systems*, pages 200–217, Newcastle, New South Wales, Australia, January 1989. Springer-Verlag. Workshops in Computing, Ed. by Professor C. J. van Rijsbergen, QA76.64.I57.

Persistent data must be contained in **secondary storage** (non-volatile disk, tape, etc.) because it is generally much larger than can be accommodated by primary storage and must persist even if power to the machine is interrupted. Secondary storage is implemented by a variety of physical media each having different addressing mechanisms. Not all items in secondary storage need to persist, but all persistent items must exist in secondary storage. The key point here is that the processor used to execute programs and manipulate the data within these programs, in general, cannot access any persistent data directly; however, this data is the most important data on a computer system.

This situation is reflected in the fact that most programming languages provide support for primary storage only. Access to persistent data is usually provided through a subsystem external to the programming language, such as a file system or database system (e.g. conventional files and directory structure). Because the definition of the persistent store is outside the programming language, the structures created are not directly usable in the programming language as are normal data structures. We believe that this dichotomy between persistent data, and volatile data should be mitigated as much as possible by presenting a uniform approach to manipulating data in a program [7, p. 49]. While the hardware is cast-in-stone to a large degree, the programming language can provide a uniform approach to addressing and accessing data.

To understand much of the mechanism we will be suggesting, it is necessary to understand the basic objectives of the system we are designing and building. First, we want to extend the notion of type safety to encompass all interactions within the system, and in particular, to interactions between a program and the persistent data it uses. Thus, it is necessary to retain the types of all persistent data so that appropriate type-checking can be done. Further, we plan to do type-checking statically. We believe that the best way of accomplishing this is to make all persistent data items definable in the programming language and to have a single programming language in the system.

Second, we want to adopt an object-oriented paradigm throughout the system. Most data will be contained in objects which define a particular set of operations that can be performed on the data. In other words, it is really objects which persist and not merely the data itself. This means it is necessary to make both data and code accessible when a persistent object is accessed. Further, objects are not normally copied in their entirety into the processes that access them. This is necessary to allow objects to be shared among processes. Thus, persistent objects must be independently accessible by multiple processes. Our mechanism for accomplishing this is to allow direct addressing of persistent objects and to make the object accessible through paging (called a segmented virtual memory scheme by Cockshott [6]). Database notions like transaction committal and nested transactions will be handled by write-through, shadow-paging and the programming language concurrency mechanism.

Third, we want to have an object which allows any type of data item to be defined and subsequently persist until explicitly removed. This will be accomplished with an instance of an `ENVIR` definition (see [4] for details), which is like a name space in other persistent systems, in that it relates names and type information with the objects created from them. One important difference between our `ENVIR` instances and name spaces is that the objects in an environment are operated on directly, whereas items in name spaces are essentially copied out of the name space, modified, and subsequently returned.

Since `ENVIR`'s are objects, one environment can contain pointers to another. Thus, our name spaces can be structured like a traditional file hierarchy but this is accomplished through programming language mechanisms. Normal programming language qualification and scope rules allow referencing both up and down the hierarchy. Hurst discusses a similar scheme in [8], but he relies on a dynamic mechanism to bind names directly with objects in a name space context (this scheme is implemented in Napier [2]). We rely on normal programming language declarations in nested blocks to bind names to objects in the block context [5]. We believe our approach is more consistent with the way programming is done in a statically-typed programming language. Having multiple name spaces introduces problems of storing data items containing pointers, as these pointers may be relative to the name space in which the data structure was created. Hence, transferring or copying information from one name space to another may be non-trivial.

At the same time, we see a necessity for supporting file/database like objects. The facilities provided by traditional files and databases cannot be incorporated into name spaces without severe problems. Abstraction, sharability, storage management and access efficiency cannot be handled well if all of the data traditionally stored in files and databases is stored in the name spaces themselves.

Intimately tied in with addressing and accessing of data is storage management of the data. How data is allocated and whether it can be garbage collected affects the form of the address for the data. Storage management is one of the most difficult aspects of any system and this is even more so in a persistent environment where user-defined objects may persist along with system-defined ones. To accommodate storage management, the programming language allows type unsafe manipulation of pointers. This is necessary to allow system programmers to design and implement objects that require specialized types of storage management. Most users would never need to do this or even be allowed to do this; nevertheless, without this facility, all storage management operations must be written in another programming language, which would make the system multilingual.

What we shall describe in this paper is the underlying storage model we propose to use to allocate and manage persistent objects such as name spaces and file objects. The storage management of objects within the secondary storage will be definable in the programming language; however, this is an abstraction level that is invisible to the normal user. Then, by structuring this storage into a hierarchy, the addresses of the persistent data within the storage will follow directly.

2. ADDRESS SPACE

The unit which we will use as the building block for persistent storage is the address space. An **address space** is a series of contiguous bytes of storage, addressed consecutively from 0 to some maximum; such addresses will be called **uniform** addresses. These address spaces are analogous to address spaces that are created for executing programs, such as data and instruction address spaces. Address spaces are not accessible to a process until some explicit action is taken, either by the user or by the system. All address spaces are accessible in a uniform way as was the objective in Multics [9]. Finally, the physical storage media (disks, tapes, etc.) can be considered to be address spaces, too. In this case, a simple computation allows one to transform a uniform address into the corresponding address required by a physical device (e.g. track and cylinder).

3. MEMORY

A **memory** is an address space resulting from instantiation of the `MEMORY` construct that will be described later. In other words, a memory is an address space with some basic structure imposed on it by the programming language. A memory may contain many objects or only one. For example, a sequential file memory contains a single sequential file object which contains information about the records stored in the file memory. In traditional systems, a file is a memory; so too is a directory, which is simply a special kind of file which points to other files.

A memory has storage management as its primary function, whereas an object corresponds to an instance of a (class-like) definition made in the programming language. However, our memories are able to be defined in the programming language. Some memories will be quite simple (e.g. a sequential file memory) and some may be complex (e.g. disk memories), depending on the kind of storage management required.

3.1. Memory Hierarchy

A disk, and the files it contains, are both memories in our model; the file memory constitutes part of the disk memory. This is an example of a two-level hierarchy of memories. Such nesting exists in other situations, too:

1. A user may be assigned a **minidisk** that contains many files, where each minidisk is treated as a disk in its own right.

2. An operating system may be assigned a **partition**, where each partition is a portion of the disk used exclusively by a particular operating system (e.g. virtual machines).

Each of these cases is a memory hierarchy which consists of three levels instead of two. This can be generalized so that the memory hierarchy can consist of many levels. Some memories, such as those for disks, contain other memories; and some, such as those for files do not.

There does not need to be any correlation between memories and storage devices. Another organization that is permitted is to have a single memory that spans several disks (i.e. treat the several disks as a single memory for storage management purposes). Such a memory, or one corresponding to a single physical storage medium, will be called a **physical memory** and will constitute the root of a memory hierarchy; a system or network will normally contain several physical memories. A leaf memory in the hierarchy would be a memory such as that for a file, which it is not designed to accommodate other memories. Figure 1 shows an example memory hierarchy.

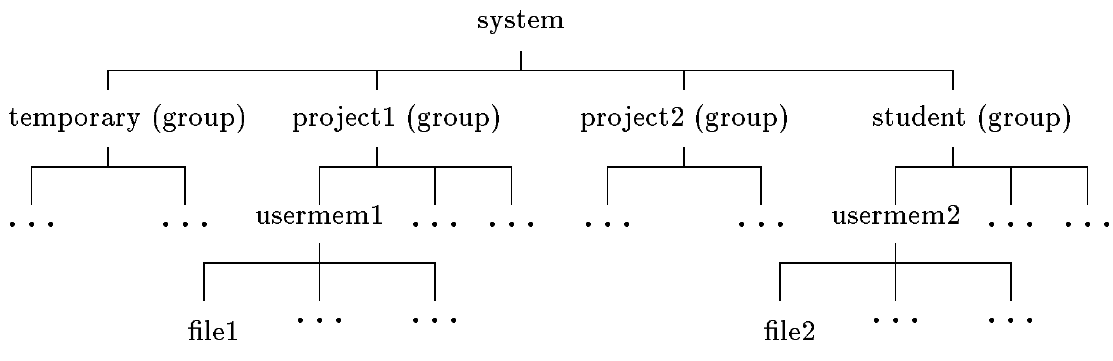

Figure 1: Memory Hierarchy

where a “group” is a submemory of the system memory and “usermem” is a submemory of a group. The system designer controls the structure of the memories that form the upper portions of the structure. Although a user will be able to create new memories by declaring files and similar objects, he/she may or may not be allowed to create memories which contain other memories (e.g. subdivide space into more minidisks) because of limitation imposed by the system designer.

Notice that the memory hierarchy does not have a large number of levels, and its organization is entirely independent of the organization of data that is contained within it.

3.2. Storage Management

An important capability of most memories, e.g. files, is that they can change size dynamically; that is, the maximum address that can be used to access data within the memory can be increased or reduced. This intrinsic property of an address space helps make storage management within the memory easier. However, it makes the storage management of the containing memory more complicated.

A memory in which repeated allocation and freeing is performed is vulnerable to fragmentation. Hence, these memories may need to use complex storage management algorithms to deal with this problem. Some form of compaction may be necessary to recover badly fragmented space; this, in turn, usually requires movement of the contained memories.

4. OBJECTS

For this paper, an **object** is an entity in a memory which can be referred to from outside the memory. The kinds of objects that can be allocated in a particular memory will depend on the definition of the memory. One important kind of object is a file; another is a name space (i.e. an `ENVIR` instance). This discussion will focus on the simpler file as it illustrates the approach. Name spaces are defined in a very similar fashion, but internally are much more complicated objects than files.

A file is an object, namely, an instance of a polymorphic class definition made in the programming language. The class definition will contain control information, such as pointers to the first and last records contained in it, as well as defining the mechanism by which the individual data structures making up the object are organized and accessed. A file differs from an entity such as an integer because its instantiation causes the creation of a separate memory within which the instance of the file class is allocated; this memory also contains the storage for all the records in the file. We refer to this memory as the **file memory**.

A program begins execution having two or possibly three memories accessible: the program memory, the stack and the heap. Simple data items needed by the program are allocated on the stack or the heap. However, these memories cannot contain other memories (such as file memories); such objects are allocated in a memory that is able to accommodate them (such as a disk memory) and a pointer to them is used instead.

Each object in a memory has a unique "identifier" through which the object can be referred to from outside the memory. Thus, the identifier of an object within a memory must remain the same as long as the object exists. In some cases, this may require that the object identifier be a pointer to a pointer to the object; such an identifier is frequently called a **handle** and it allows objects within a memory to be moved around without affecting the identifiers to the object. This might be the case, for example, for a disk memory where files are reorganized occasionally to improve speed of access of information on the disk.

4.1. UNDERLYING OBJECTS

As stated, a file is contained in a memory which can expand and contract. The file memory is not necessarily stored as a contiguous area on the disk but as a series of extents (pieces) scattered throughout the disk memory. Pointers to these extents are contained in an object in the disk memory, usually called a **file descriptor**. We say that the file descriptor object **underlies** the file memory. Thus, a file consists of three levels:

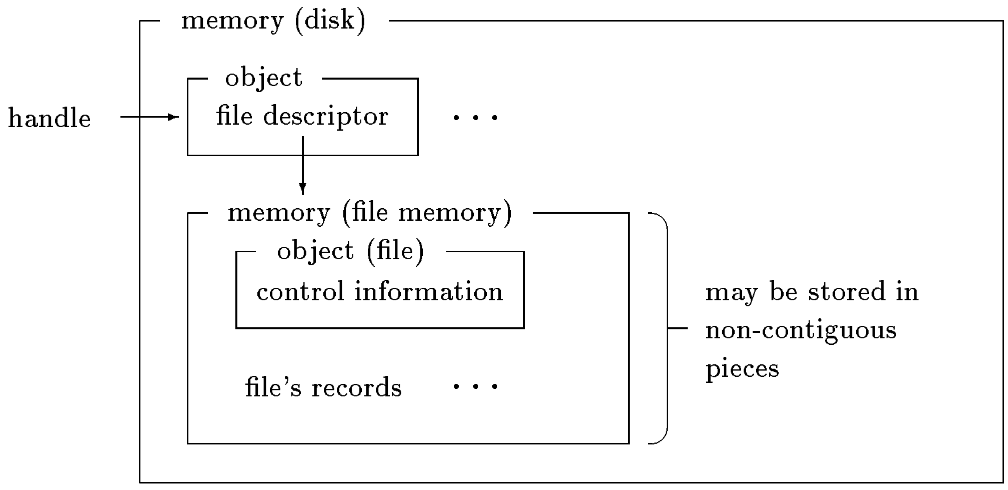

Figure 2: Memory Structure

1. the file object itself
2. the memory in which it is contained
3. an object underlying this memory (and hence underlying the original object)

The latter two are allocated in the next higher level memory (i.e. the disk). This structure is illustrated in Figure 2.

This is the model we use generally for all memories – each has an associated underlying object which is (usually) allocated in the next higher level memory in the memory hierarchy. In particular, there will be an object underlying the memory corresponding to a physical storage device; in the case of a disk memory, for example, this object interacts directly with the device hardware. This object is the disk drive and is not allocated in any memory.

4.2. Object Identifiers

An object has a unique identifier within the memory in which it is allocated, but this is clearly relative to that specific memory. To obtain a system-wide unique identifier for the object we need to specify, as well, the identifier of the memory in which it is allocated. This is done by giving the identifier of the object underlying the memory. Similarly, the identifier of the underlying object will involve the identifier of the memory in which it is allocated, etc. Thus, we build up a system-wide identifier for an object by giving in order the sequence of identifiers of all underlying objects in which the object lies, plus the identifier of the desired object in its memory. Such a multi-component identifier is called an **object identifier**. It has one component for each level in the memory hierarchy containing the object. Hence, each memory in the memory hierarchy plays an active role in the formation of an object identifier. This identifier is the internal analogue of a qualified name in a programming language (e.g. x.y.z).

4.3. Dismountable and Distributed Memories

There is always a one-to-one correspondence between memories and underlying objects. This permits the identifier of the underlying object to be used as the unique means of identifying the corresponding memory.

For a physical memory, which includes dismountable and distributed memories, this structure is preserved. However, in this case, the underlying object exists in no memory and hence cannot be assigned an identifier relative to it. Instead, the object must be assigned an identifier that makes it unique within the system/network. This will be called its **memory identifier**. (The underlying object is the analogue of block zero of a disk, which contains the location of its root-level file directory.) This identifier is determined at the time the memory is created (initialized) and will probably include as part of its identifier the unique identifier for the system.

Each device, such as a disk drive, is controlled by a volatile object on the machine in which the device is installed. The object is volatile because it disappears when the machine is powered off. There is a network-wide table that relates the memory identifiers of all physical memories currently available on the network, to the volatile objects on which they are mounted.

When the system attempts to decode (dereference) an object identifier, it uses the memory identifier to find the volatile object (drive) on which the memory is currently located. Thus, from the memory identifier, the system infers the appropriate object to use to access the memory.

5. MAKING A MEMORY ACCESSIBLE

Because most memories are in secondary storage, they are not directly accessible by the hardware; hence, the objects within them are not accessible either. Conceptually, to make an object accessible, its containing memory is copied into primary storage. Clearly, copying the entire memory may not be possible if the size of the memory exceeds that of primary storage, or it may be inefficient even if primary storage is large enough and it makes sharing difficult. **Paging** offers a solution to this problem by allowing a partial copying, and it makes the copying invisible and under system control. From the user's standpoint, the object is accessed directly.

5.1. Representative

When a memory is made accessible by a program, the system will create an object in primary storage to keep track of the location of the partial copy (or set of pages). This object and the partial copy are analogous to an object and its overlying memory, except these are created in primary storage. In effect, we are creating a two level memory hierarchy in primary storage analogous to the memory hierarchy created in secondary storage. This primary storage analogue of the object underlying a memory we call a **representative**. It is an extension of the underlying object – the representative uses information in the underlying object to locate the memory, but it must also keep information about the copy in primary storage.

Besides holding data locating the copy of the memory, the representative also contains procedures defining how and when to make the memory accessible. The representative will operate in cooperation with the particular kind of storage management provided by the memory. It might bring all the pages into primary storage and hence simply perform swapping, or, it might perform prefetching and purge behind of pages so that there is a window of addressability that moves through the file as it is accessed. For a data base, the representative can keep certain areas of the primary index and data area constantly accessible and let other areas of the index and data be made accessible on a demand basis.

Only one representative for a particular memory is created, and it is used by all the accessors on that system. This property is essential because allowing duplicate copies (complete or partial) in primary storage is storage inefficient and causes consistency problems during concurrent access. To accomplish this requires that the system maintain information about which memories are being accessed and create a new representative only if one does not already exist; otherwise the existing representative is used.

We are currently trying to extend the role of the representative to deal with remote access to memories across a network. What is necessary is to have a representative created in the file server for the memory that is being accessed and a representative created on each machine that is accessing the memory. The representatives can then communicate with each other to manage paging/swapping and sharing/updating across the network.

6. FORMATION OF A MEMORY HIERARCHY ADDRESS

When an object is declared, both the instance and an address to refer to it are created. The address that is created may only have to be a simple identifier if it is only used within the memory in which the object is allocated. However, when a file is declared, the identifier that is returned for it must, in general, be able to be dereferenced in a memory other than the one it is immediately relative to. Therefore, the identifier of such an object must include a complete set of components from the root of the memory hierarchy to the newly allocated file. The following discusses the structure of the address and how it is constructed.

6.1. Component Value

Each component value of an object identifier may be the displacement from the beginning of the memory to the storage for the object. This value is determined at execution time when the object is allocated. The value returned from the allocation is the identifier of that object in the containing memory.

However, the value returned from allocation does not have to be the actual address of the item in the memory, it may be a handle. For example, a simple subscript value to an array of object pointers (to which handles point) might be returned. Thus, only this smaller amount of information, which is an **encoded** form of the location of the object, must be stored in the identifier. This is an important point, as it means that a component value need not depend on the size of the object's memory that it identifies.

Because a component may be encoded, it is necessary to decode the component after the memory to which it applies has been made accessible. Thus, each memory must contain a routine that transforms the component value into the address of the desired object within the memory. This routine would be called as part of dereferencing to actually locate an object in a memory.

7. MEMORY HIERARCHY ADDRESS RESOLUTION

An object identifier cannot be used by the processor to access an object as it uses normal addresses of entities in primary storage. In order to do this, the memory containing the object must first be made accessible. Making the memory that contains an object accessible involves dereferencing the object identifier, and making all of the lower level memories accessible, if necessary. Each successive component (except the last) of an object identifier is used to locate the object underlying a memory and to make it accessible (by paging or swapping). Then the next component is able to be used, since it is relative to the memory just made accessible.

Each component in the object identifier indicates an object in an overlying memory. Thus, in traversing the memory hierarchy, objects in several memories may have to be made accessible, and hence several representatives may have to be created, one for each object. It is the representative and the underlying object that make the overlying memory accessible to the processor.

The first step in resolving an object identifier is to locate the memory whose object identifier is given by the first component of the object identifier. The representative of all objects capable of holding physical memories are examined to locate the one holding the desired memory. This step may fail if no appropriate object can be found. Once found, the next component is decoded by passing it to the decode routine for this memory, which must be accessible because it is mounted. The decode routine returns the identifier of the object specified in the encoded component. This object underlies the next memory. The above process is repeated on the next component for the memory overlying the current object. However, this memory may or may not be accessible, and if not accessible, it must be made accessible to decode the next component.

Notice that the levels of the memory hierarchy nearest the root will normally remain active and hence they will already have representatives created. Thus, the first part of the object identifier will likely be able to be resolved quickly. Only for the later components of the object identifier will new representatives need to be created. All intermediate memories accessed must remain accessible until the access to the object is finished. This is because each memory is dependent on the one in which it is contained to be able to directly access its underlying object and to allow access to the blocks of the overlying memory.

7.1. Active Memory Tree

As has been mentioned already, only one representative is created for an object no matter how many accesses are made to it. To do this the system must have some way, during identifier resolution, of determining if a representative for an object already exists, and if it does then it uses the existing representative.

This is accomplished through the standard technique of maintaining a list (or tree) of the active memories in primary storage, called the **active memory tree**. The active memory tree is a hierarchy built in primary storage which contains the identifiers of all active objects. The structure of this tree corresponds to the structure of the memory hierarchy, but it is only the small subset of the memory hierarchy that is currently active. Each level in the tree represents a level in the memory hierarchy. At each level, there is a list of all the representatives for objects that are active. The size of the active memory tree is a function of the number of accessed objects.

The resolution of an object identifier may produce several entries in the active memory tree. Resolution yields the address of the representative of the last memory in the sequence.

8. PROGRAMMING LANGUAGE CONSTRUCTS

This section gives a simplified description of the programming language constructs that we intend to use to define the memories and objects described above (see [3] for a more detailed description).

8.1. `MEMORY` Construct

The `MEMORY` construct is a class-like construct defining a particular kind of memory. For example, a definition for a sequential file memory might look like:

```
FileDesc MEMORY SeqMemory(t : TYPE) FileDesc is the superclass for SeqMemory
   VAR RecordStart, RecordEnd : REF t

   PROC append(r : t) ... allocate & add a new record
   PROC rewrite ... destroy previous file contents & reinitialize
   PROC alloc(s : INT) RETURNS COMPONENT ... allocate storage for object
```

```
   PROC free(a : ADDRESS) ... free storage for object
   PROC decode(a : COMPONENT) RETURNS a : ADDRESS ... decode component
END MEMORY
```

Each kind of memory may provide different storage management primitives appropriate for the kind of object (or objects) to be allocated within it; for example, a disk memory would likely provide one kind of storage management for file memories and another for file descriptors.

Each memory must provide routines to allocate and free storage for objects created in the memory. This scheme can be further generalized by having the definition of an object specify which routines in the memory are used for its allocation and freeing. This allows specific allocation and freeing operation for different kinds of objects if each kind requires different storage management. As well, each memory must have a routine that decodes the object identifier components that are returned by the allocation routines. This routine can be designated by having a special name, such as `decode`.

Since a memory must do storage management, it must be able to manipulate and assign pointers as integers. Although this violates our objective of type safety, it appears to be a worthwhile sacrifice in order to be able to define memories in the programming language. Such violations are allowed only in memory definitions and these definitions will usually be written by system programmers and so type safety will not be ordinarily compromised.

8.2. Object Definition

The `OBJECT` construct is a class-like construct that defines a particular type of object. Because an object is allocated in a memory, the object definition must specify the type of memory to be created in. This is specified by making the object a subclass of the memory in which it is to be allocated. For example, a definition for a sequential file object might look like:

```
SeqMemory OBJECT SeqFile(t : TYPE) : SeqMemory(t)
                                     ALLOCATE alloc FREE free
   VAR FirstRecord, LastRecord, CurrRecord : REF t

   PROC read RETURNS REF t ... read a record
   PROC write(r : t) ... write a record
   ...
END MEMORY

VAR f : SeqFile(INT)
```

For the declaration of `f`, the compiler will generate the appropriate code to create an instance of the superclass `SeqMemory` and then a call to `alloc` to obtain storage for a `SeqFile` object and an object identifier component. When the block containing `f` is released, an appropriate call to `free` will be made.

The mechanism explained above, by which a memory and the object it contains is made accessible, makes the data items in them just ordinary variables which can be accessed and assigned to as usual. This is unlike traditional code for operating on files/databases which is written so that data from them is explicitly brought into primary storage. In this design, this distinction is largely blurred so that a programmer has a single uniform paradigm in which to work, that is, defining a memory or object is the same as defining a class. For example, updating a single variable in a traditional file involves explicitly reading in the block that contains the variable, changing it, and explicitly writing the block back to the file. In our case, only a simple assignment statement will modify any variable defined in the memory or object.

8.3. Object Underlying a Memory

The file descriptor contains the list of extents for the file memory and also routines that can be used to extend or reduce the size of the file memory and would look like:

```
OBJECT FileDesc
   VAR extents : [15] ExtentDescription

   PROC extend(size : INT) ...
   PROC reduce(size : INT) ...
END OBJECT
```

8.4. Representative

To make a `SeqMemory` accessible, a representative for its underlying object is created. This representative handles paging (or swapping) of the memory. The definition of the representative forms part of the definition of the underlying object so that its code can refer to the data items in the underlying object, as in:

```
OBJECT FileDesc
   ...
   REPRESENTATIVE no name, created by the system
      VAR ActivePages : [?] INT

      PROC PageFault(ADDRESS : INT) ...
      PROC PageOut(ADDRESS : INT)
   END REPRESENTATIVE
   ...
END OBJECT
```

The representative is created implicitly by the system on the first access to the memory and persists across all accessors. The system will call the routines in the representative when page faults occur or when pages must be paged out.

8.5. ENVIR

An `ENVIR` definition defines a new name space which allows types to be defined and any data items to be declared (see [4] for details). All types of definitions can be nested in an `ENVIR` definition which in turn could have further nested definitions, for example:

```
ENVIR eee     define an environment type
   CLASS ccc ...
   ENVIR fff ...
   VAR i : INT
   PROC p( ... ) ...
END ENVIR
```

```
VAR e : eee   create an environment
```

Like a class, it is possible to refer to data items in `e` from outside the `ENVIR` instance, for example:

```
e.i ← 2
e.p( ... )
```

What makes an environment instance different from a conventional block is that it can be used as a context for compilation and it can be dynamically changed, by adding or removing symbols from it, or by changing an existing symbol's definition. To accomplish this, an environment is made self defining by creating its own symbol table and retaining it along with the symbol tables of any nested definitions. As well, an environment has its own code area to store code derived from the definitions defined in its symbol table. Thus, there is a one-to-one relationship between an environment and its symbol table (unlike an instance of a class). Because of this relationship, we are able to assure that any change made to an environment symbol table will cause corresponding changes to the environment itself.

Because each environment has its own symbol table, it does not have to remain identical to its original defining type. Unlike a normal type which can have multiple instances, of which each one is identical in structure, each instance of an `ENVIR` definition can be extended independently, being described precisely by its symbol table. Hence, an environment is unique in that it provides a context for compilation (the symbol tables and its structure) and behaves like a data area. The data area, symbol table and code area allocated for an environment instance are created as separate memories. This mitigates much of the storage management problems associated with making dynamic additions and deletions of variable sized items within each.

Internally, storage management is aided by having the data area memory use handles to refer to items in the environment. Hence, it is possible to move items in the environment data area without affecting external addresses to these items. This allows garbage collection to be performed on the environment data area.

A complex `ENVIR` definition can be constructed that defines an environment for a user. It is constructed using the same basic building blocks as for a file, as in:

```
UserDataArea ENVIR User( ... )
    ...
UserDesc MEMORY UserDataArea( ... )
    ...
UserMiniDisk OBJECT UserDesc( ... )
    ...
MiniDisk MEMORY UserMiniDisk( ... )
    ...
```

Instantiation of the environment definition `User`, causes the implicit creation of the three memories to contain the information for that user's working environment. `UserDataArea` is the memory for the data area that contains storage for the simple variables that are declared in the environment. (Since the symbol table and code area are managed by the system, there is no explicit superclass for them; they are created implicitly.) `UserDesc` is the underlying object that maps the `UserDataArea` memory. `UserMiniDisk` is the memory that contains the blocks of storage for `UserDataArea` and the blocks of storage for any other memories allocated in environment `User`.

Entities declared in instances of environment `User`, that create a memory, must create it separately from the `UserDataArea`. This is because `UserDataArea` is not meant to contain other memories. It is meant to manage the storage for the environment which contains all the simple variables. The storage for objects such as files comes from the user's minidisk which also contains the environment data area, symbol table and code area.

When an object like `SeqFile` is declared in `User` instance, the compiler can determine that it cannot be allocated in `UserDataArea` from its definition and `User`'s definition. The compiler then follows the superclass chain and discerns that a `SeqFile` can be allocated in `UserMiniDisk`, and hence creates a pointer to the `SeqFile` in `UserDataArea` and creates the file in `UserMiniDisk`. This is analogous to the situation of a procedure that allocates a local `SeqFile`. The `SeqFile` is not allocated on the stack with the procedure's local variables but instead is allocated in the memory in which the program executes, and points to the file from the stack. This mechanism allows the underlying memory for an object to be determined implicitly. If necessary, the underlying memory may be explicitly specified to allocate the object in a memory other than the users.

9. CODE SEGMENTS

Our language is block structured and allows separate compilation units at arbitrary levels in this structure.

```
OBJECT SeqFile ... COMPILABLE
   ...
   PROC read ... COMPILABLE
      ...
      PROC local ... COMPILABLE
      ...
   PROC write ...
   ...
```

Here each entity that has a `COMPILABLE` clause is eligible to be compiled separately, and components with no compilation clause are compiled as part of the node that contains them. In the above example, `write` is compiled as part of the compilation of `SeqFile`. As well, it is possible to define precisely the location of object code generated by the compilation of each compilation unit. By introducing a definitional-time declaration of a code area, called a **segment**, it is possible to have each compilation unit specify where its code will be placed, as in:

```
SEGMENT m, n     definitional-time declaration

OBJECT SeqFile ... COMPILABLE m   code is placed in segment m
   ...
   PROC read ... COMPILABLE n     code is placed in segment n
```

Object code is executed directly from segments; hence, it is not necessarily to link object code together to form the executable equivalent of a `PROGRAM` in Pascal.

The same scheme presented for managing data objects in memories is used for managing segments. A code image is the executable code produced by the compilation of a separate compilation unit of the programming language. A segment can contain several code images, just as disk memory can contain many objects (i.e. instances of definitions). Each code image will have an object identifier that can be used to refer to it.

9.1. Accessing Segments

Each segment is a memory, and hence must be made accessible before it can be accessed. This accessing will create a representative to page the segment that is accessed by the processor. A segment is made accessible when the context in which it is declared is entered. When an object is made accessible, the segment that contains the code to manipulate the memory must also be made accessible. Thus, the segment is made available as a result of the access to an object. To accomplish this, the object identifier of the code image is associated with the memory (as a field in the underlying object for the memory).

10. PROCESSOR ADDRESS SPACES

A **processor address space** is an address space that the processor makes use of in order to execute a program, such as the code segment, data address space, stack address space, etc. There are a fixed number of these (as defined by the processor). A memory address space may be larger than a processor address space in which case only a portion of the memory can be accessed directly.

The design presented here requires that the processor be capable of having several active address spaces. The idea of multiple active address spaces was developed in the Multics system. In the case of micro and mini computer systems, multiple active identifier spaces were introduced because the address space size on these systems severely restricted the amount of accessible memory. Hence, these systems were forced into multiple active address spaces not because of the desire to solve storage management problems but simply to be able to increase the amount of data that could be addressed.

Our system requires several address spaces to be directly accessible by the processor:

1. segment address space - memory for the code for the current object
2. stack address space - memory for the local variables of procedures and for parameters
3. heap address space - memory for the dynamically created objects
4. data address space - memory containing the current object
5. temporary data address space - memory for the objects that must be accessed temporarily, such as parameters in an inter-task call, or data moved from one memory into another memory

All the address spaces must be explicitly selectable in the identifier portion of an instruction. For example, the segment address space may be accessed explicitly because it may contain constants.

These five active address spaces will be accessed through five hardware registers on the processor. These hardware registers point to the representatives for the address space instead of to the actual data structure for a page table. This implies that the hardware is aware of the structure of a representative and not just of a page table.

10.1. Changing of Processor Address Spaces

During a procedure call, some or all of these address spaces may be changed. The conditions that cause each identifier space to change are situations such as the following:

1. The segment changes when a call is made to a routine in a different segment.
2. The stack and heap change during a call to a routine in another task.

3. The data area changes during a call to a routine contained in an object in another memory. For example, the declaration:

```
VAR f : SeqFile(INT)
```

creates an object which creates a new memory. `SeqMemory` and `SeqFile` and all their routines will likely be compiled in one segment. Based on the definitions of `SeqFile` and `SeqMemory`, the compiler can determine which of the processor address spaces contain the various variables and routines that are used in the statements; and, if a variable or routine lies in another accessible address space that is not currently a processor address space, how this address space can be accessed.

10.2. Implications of Processor Address Spaces

During a call to a procedure, if a data item is passed by address as an argument or when a pointer is passed as an argument, it may be necessary to expand the address to identify the memory which it is relative to; one possible way of doing this is to pass the object identifier of the argument. However, constructing the complete object identifier dynamically, and dereferencing it in the called routine would be expensive. A simpler solution is to pass both the simple address of the data and the identifier of the representative of the object underlying the memory. This kind of variable or parameter is called a `REPREF`.

While it is conceivable to implement all such addresses using `REPREF`s, it is undesirable for routines that operate within the same memory. These can accept parameters by simple address which are known to be relative to the active memory.

Pointers to data internal to an object that are returned as results by public routines must be `REPREF`s, because the pointer is relative to an address space different from the caller's. If a simple `REF` is returned by the routine, it is the caller's code that constructs the `REPREF` using the object's representative pointer. If an actual `REPREF` pointer is returned, the `REPREF` is constructed by the callee's code.

When a `REPREF` is passed as an argument to a routine expecting a `REF`, a dynamic check is performed to assure that the `REPREF` is identifying the appropriate memory. If it is not the same, an exception is raised. This happens for:

```
OBJECT xxx
   PROC yyy( p : REF m )
      ...
   ...

VAR x : xxx
VAR y : REPREF m
x.p(y)
```

While `y` can point to data in different address spaces, it must point to `x`'s when it is passed to a routine in `x`.

11. CONCLUSION

We believe this design provides a consistent approach for supporting access to secondary storage. It makes this storage available in the programming language in almost the same way as primary storage. Users can define its structure (within the constraints imposed by the system definitions), which objects appear in memory, how the storage in the memory is managed to contain these objects, and how the memory is made accessible to the processor. A memory interacts with the system in two ways: first, by interacting with the compiler and run-time system to allocate and free objects in a memory; secondly, by interacting with the addressing system through encoding and decoding of the component of an object identifier to a contained object.

Unlike traditional flat addressing schemes, the addressing system that results from this design specifies: a straight forward transformation for locating information, unique identifiers that can be reused if the memory that generated them can reused them, some amount of transportability through dismountable memories, and incorporation of devices as a direct consequence of the underlying-object/memory pairing. Finally, this design can be implemented, as there exists at least one processor, the Intel 386, with sufficient hardware capability to support multiple active address spaces.

REFERENCES

1. Albano, A., Cardelli, L., and Orsini, R. "Galileo: A Strongly-Typed, Interactive Conceptual Language". *ACM Trans. Database Syst.*, 10(2):230–260, June 1985.

2. Atkinson, M. P. and Morrison, R. "Types, Binding and Parameters in a Persistent Environment". In *Workshop on Persistent Object Systems: their design, implementation and use*, volume PPRR 16, pages 1–24, Appin, Scotland, Aug. 1985. Universities of Glasgow and St. Andrews, Scotland.

3. Buhr, P. A. and Zarnke, C. R. "A Design for Integration of Files into a Strongly Typed Programming Language". In *Proceedings IEEE Computer Society 1986 International Conference on Computer Languages*, pages 190–200, Miami, Florida, U.S.A, Oct. 1986.

4. Buhr, P. A. and Zarnke, C. R. "Persistence in an Environment for a Statically-Typed Programming Language". In *Workshop on Persistent Object Systems: their design, implementation and use*, volume PPRR 42, pages 317–336, Appin, Scotland, Aug. 1987. Universities of Glasgow and St. Andrews, Scotland.

5. Buhr, P. A. and Zarnke, C. R. "Nesting in an Object Oriented Language is NOT for the Birds". In Gjessing, S. and Nygaard, K., editors, *Proceedings of the European Conference on Object Oriented Programming*, volume 322, pages 128–145, Oslo, Norway, Aug. 1988. Springer-Verlag. Lecture Notes in Computer Science, Ed. by G. Goos and J. Hartmanis.

6. Cockshott, W. P. "Addressing Mechanisms and Persistent Programming". In *Workshop on Persistent Object Systems: their design, implementation and use*, volume PPRR 16, pages 369–389, Appin, Scotland, Aug. 1985. Universities of Glasgow and St. Andrews, Scotland.

7. Cockshott, W. P., Atkinson, M. P., Chisholm, K. J., Bailey, P. J., and Morrison, R. "Persistent Object Management System". *Software – Practice and Experience*, 14(1):49–71, 1984.

8. Hurst, A. J. "A Context Sensitive Addressing Model". Technical Report PPRR-27-87, University of Glasgow and St. Andrews, Scotland, 1987.

9. Organick, E. I. *The Multics System.* The MIT Press, Cambridge, Massachusetts, 1972.

10. Pitts, D. V. and P., D. "Object Memory and Storage Management in the *Clouds* Kernel". *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 10–17, June 1988.

11. "Taxis'84: Selected Papers". Technical Report CSRG-160, University of Toronto, Toronto, Ontario, Canada, June 1984. Ed. by Brian Nixon.

12. Wileden, J., Wolf, A. L., Fisher, C. D., and Tarr, P. L. "PGRAPHITE: An Experiment in Persistent Typed Object Management". In *Proceedings of SIGSOFT'88: Third Symposium on Software Development Environments*, 1988. to appear.