MIT/LCS/TR-209

A MACHINE ARCHITECTURE TO SUPPORT AN OBJECT-ORIENTED LANGUAGE

Alan Snyder

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661

MIT/LCS/TR-209

A Machine Architecture to Support an Object-Oriented Language

by

Alan Snyder

March 1979

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract N00014-75-C-0661.

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge Massachusetts 02139

A Machine Architecture to Support an Object-Oriented Language

by

Alan Snyder

Submitted to the Department of Electrical Engineering and Computer Science
on March 9, 1979 in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Abstract

In object-oriented languages (e.g., LISP, Simula, and CLU), all (or most) data objects used by a program are implicitly allocated from a free-storage area and are accessed via fixed-size references. The storage for an object is automatically reclaimed (garbage collected) when the object is no longer accessible to the program.

This thesis presents the design of a computer system that directly supports an object-oriented machine language. The machine provides a single, large universe of objects shared by multiple processes. The design uses expected future technologies (fast-access secondary storage devices and inexpensive processors) to satisfy the goals of good performance and a simple, modular system organization.

Automatic storage reclamation is performed primarily using reference counts. The proposed reference count implementation reduces the time overhead of automatic storage reclamation and allows most reclamation processing to be performed in parallel with normal computation. In addition, the reference count scheme can be used in a multiprocessor configuration without introducing complex synchronization problems.

A proposed implementation of the machine is described in terms of a number of specialized processor modules communicating via messages. Multiple processors are used to improve performance and to achieve a more modular system structure.

Thesis Supervisor: Barbara H. Liskov
Title: Associate Professor of Computer Science and Engineering

Keywords: language-oriented machine architecture, object-oriented programming languages, garbage collection, reference counts

Acknowledgments

I would like to thank Barbara Liskov for her efforts in supervising this thesis. I would also like to thank my readers, Liba Svobodova and Jack Dennis, for carefully reading the thesis and providing helpful comments. Thanks are due also to Eliot Moss, who read a draft of the thesis and pointed out a number of areas needing clarification.

CONTENTS

FIGURES

1. Introduction

1.1 Motivation

The design of computers is strongly influenced by the characteristics of available technology. Until recently, computers have been designed under the constraint that processing hardware is expensive. The resulting desire to minimize hardware cost has had a number of unfortunate effects.

One effect is that conventional machines generally provide a rather low level machine language, thus encouraging the use of programming languages with similar low level (or "machine oriented") semantics. Although better programming languages have been developed, their implementation on conventional machines is often excessively inefficient.

Another effect is that individual processors are multiplexed to perform many different functions. For example, a single processor is often used to interpret user processes, implement the virtual memory, and control I/O devices. This multiplexing is supported by a complex, interrupt-driven operating system, characterized by considerable interactions among its various components. Such complex systems are difficult to understand or verify and are likely to be unreliable.

The cost of hardware is continually decreasing and the significant cost of software has become more and more apparent. Therefore, we believe it is appropriate to consider how hardware technology can be used to implement better programming languages and to reduce the complexity of computer systems.

1.2 Goals

This thesis presents the design of a new computer system that efficiently supports a single, large universe of *objects*. Our notion of objects appears in a number of programming languages, such as LISP [36], Simula [6], and CLU [22]. The essential characteristics of such *object-oriented* languages and their advantages over traditional *value-oriented* languages are described in detail in the next chapter. We note at this point, however, that an important implementation implication of object-oriented languages is the use of automatic storage reclamation (garbage collection).

Unlike most implementations of object-oriented languages on conventional machines, which provide a separate and usually small space of objects for each process, our computer system instead provides a single, very large space of objects shared by all of the processes in the system. This space of objects would include not only temporary objects used during the execution of programs, but also the "permanent" procedures and data normally stored in a file system.

Having a single, large *universe* of objects means that there is no distinction between objects that are local to a process and those that are stored in the permanent file system. There are no artificial barriers between different processes or between processes and the file system. Objects in the file system can be accessed directly, with no restrictions on the types of objects that may be permanently stored and no need for conversions.

The primary goal of this thesis has been to design a machine that effectively supports a large universe of objects. A second goal has been to minimize the complexity of the design. We want the machine to have a simple and modular structure, in comparison with conventional machines and operating system kernels that provide similar functions. To accomplish these goals, we have made two assumptions about expected future technology:

The first assumption is that processors are sufficiently inexpensive that we can use a number of processors where one is used today. We use multiple processors to obtain greater modularity in the implementation of the system, as compared to current systems where processors are a scarce resource and must be multiplexed to serve many different functions. In addition, we use multiple processors to improve the performance of the system, particularly the implementation of automatic storage reclamation. Our goal is to increase the throughput of the system, although perhaps at the expense of decreased utilization of resources.

The second assumption is the existence of fast-access secondary storage devices that can be used for file storage. We are envisioning an access time on the order of 100 microseconds (compared to 10 milliseconds for current disks). Such devices are used to obtain good multi-level memory performance without introducing undue complexity. This assumption is motivated by the expected small average size of objects, based on measurements of existing programs; it is not needed if the objects supported by the system are mostly large objects, similar to pages or segments in current virtual memory systems.[1]

[1]Aside from a fast access time, we are not assuming any special properties of the secondary storage devices. For example, we do not assume the ability to scan the entire secondary storage in one access time, an ability provided by charge-coupled devices.

1.3 Background

We are proposing a single large space of objects whose size is similar to that of current file systems. Thus, the universe of objects will be implemented by a multi-level memory system. Current architectures cannot efficiently support such a large space of objects. The major problems involve the performance of the multi-level memory system.

Current secondary storage devices are characterized by access times that are many orders of magnitude longer than the access time of primary storage. To achieve reasonable performance in a multi-level memory system, it is thus necessary that the rate of access to secondary storage be quite low. To keep the access rate low, it is necessary to transfer a fairly large amount of *useful* data on each transfer from secondary storage. (Transferring a larger amount of data than is actually needed to fulfill a request is beneficial only if some of that additional data will be needed in the near future.)

Page sizes in current multi-level memory systems range from 128 words to 4096 words. However, programming-language objects tend to be quite small. A median size of under 20 words is not unlikely [3], and we have measured programs whose average object size is only three words (see Section 6.8). Thus, to efficiently implement an object-oriented storage model using current secondary storage devices, it is necessary to group related objects together in some manner and transfer them as a unit between primary and secondary storage.

Conventional implementations of object-oriented languages group related objects together implicitly through the use of small object spaces. Each space of objects contains the objects needed by a single process. Thus, the space can be broken into a small number of pages all of which contain objects that are likely to be used by the process. The use of a compacting garbage collector tends to minimize the number of pages used by each space of objects, thus reducing the working sets of the processes [10]. Furthermore, since each small space of objects is garbage collected separately, the time required for garbage collection is (in most cases) tolerable.

These techniques are not easily extended to the case of a single, very large space of objects. Here, some additional grouping of objects is needed to take the place (for performance reasons) of the small object spaces in conventional systems. The mechanism for grouping objects can be either explicit (the groups are constructed by the user) or implicit (the groups are constructed automatically by the system). In addition, it must be possible to perform garbage collection on individual groups of objects. A garbage collection of an entire multi-level memory system would be quite slow. Even if the garbage collection were performed concurrently with normal system operation, it would tie up the secondary storage devices and significantly reduce the performance of the system. Furthermore, unless garbage collection is performed reasonably frequently, it will have little effect on the working sets of the active processes.

A system with these characteristics has been designed by Bishop [7]. In his system, objects are grouped into areas, which are explicitly created and manipulated by users. Objects may be explicitly assigned to areas by the programmer. However, explicit assignment of objects to areas is not necessary; a mechanism is provided that will automatically move objects to the "proper" areas. In addition, Bishop's system includes a compacting garbage collector that works on individual areas or groups of related areas.

We believe that Bishop's approach to the implementation of a large universe of objects is the correct one, given current secondary storage devices. However, his system is complex and its user interface (with areas) is more complex than necessary. We believe that expected future technology will allow a very large universe of objects to be implemented without any notion of areas and in a relatively simple and modular way.

1.4 Our Approach

As stated above, we are making two assumptions about expected future technology. The first assumption, inexpensive processors, is fairly safe. LSI processors are being introduced today that are comparable in power to minicomputers of a few years ago. It is widely predicted that LSI processors equivalent to current mainframes will be developed in the next decade. The cost of these processors will be quite low compared to the total cost of a computer system. As a result, it will be feasible to use many processors in a single system.

The other assumption, fast-access file storage devices, is more questionable. The access time figure of 100 microseconds is within the predicted range for charge-coupled devices and electronic beam memories [33]. Electronic beam memories, however, are still in the research stage. Commercial charge-coupled devices are beginning to appear; however, the major question here is whether their cost will ever be sufficiently low to allow them to replace disks. Current projections [25] show the cost of CCDs approaching the current cost of disk memories. However, these projections also show the cost of disk memories decreasing, so that disk memories will still be an order of magnitude less expensive than CCD memories. Thus, although it is difficult to predict that CCDs will *completely* replace disks in the near future, they can be expected to be used in reasonably large quantities for file storage.

The assumed existence of fast-access secondary storage devices means that it is possible to swap individual objects between primary and secondary storage with acceptable performance. The expected decrease in the average amount of information transferred on each secondary storage access is about a factor of 100 (10 word objects vs. 1000 word pages). We would expect, therefore, that the rate of access to secondary storage would increase by at most a factor of 100.[2] However, the expected decrease in the access time compared to current secondary storage devices is also about a factor of 100. Thus, it is reasonable to predict that the performance of a system that swapped individual objects to and from fast-access secondary storage devices would be no worse than the performance of current multi-level memory systems.

In many current systems, the time required to perform a secondary storage transfer is rivaled by the time spent by the processor in handling the page fault, locating the page in secondary storage, scheduling the transfer, switching to another process, etc. Unless this overhead can be reduced, full advantage could not be taken of the improved secondary storage access times postulated. We do not believe that a large overhead is inherently necessary. Much of the overhead in current systems represents attempts at optimization that are appropriate only in the context of a very long secondary storage delay. Other aspects, we believe, can be reduced by proper design.

We also note that a full factor of 100 improvement in the secondary storage access time may not be needed to achieve good performance. For one thing, swapping individual objects rather than pages can allow a given amount of primary storage to capture a greater portion of the working sets of executing processes, thus reducing the secondary storage access rate. In addition, the rapidly falling cost of primary storage will allow larger amounts of primary storage to be used, which can also reduce the secondary storage access rate.

[2]It is possible for the number of secondary storage accesses to increase by more than a factor of n when the page size is cut by a factor of n [24, 16]. However, we do not believe this anomaly to be a serious problem in practice.

However, neither of these effects reduce the number of secondary storage accesses that occur when a program or data base is initially accessed. To be conservative, we will continue to assume the factor of 100 improvement.

If individual objects are swapped, then there is no need to group objects together for performance reasons. Thus, there is no need for areas or an automatic grouping mechanism. Similarly, there is no need to have a compacting garbage collector to improve locality. (A compacting garbage collector might still be useful for reducing storage fragmentation.) Instead, we can use reference counts as our primary means of automatic storage reclamation. Infrequent garbage collection can be used to reclaim inaccessible objects that cannot be detected using reference counts.

1.5 Overview

The initial chapters present an overview of the architecture and design of the machine. In Chapter 2, we describe the object-oriented storage model in detail and discuss the advantages and implementation implications of this model. In Chapter 3, we describe the visible architecture of the machine and the philosophy of its design. In Chapter 4, we give an overview of the implementation of the machine. The system is presented as consisting of two major modules, one implementing the object-oriented memory, the other implementing processes. Our approach to the implementation of multiple processes is briefly explained.

The remainder of the thesis concentrates on the memory module. The next three chapters discuss specific issues in the implementation of objects. In Chapter 5, we describe the implementation of object references and compare our implementation to previous work. In Chapter 6, we consider the implementation of automatic storage reclamation. In Chapter 7, we discuss storage allocation.

The following chapter, Chapter 8, presents a specific memory module design, consisting of a number of hardware modules that communicate by passing messages. The chapter pays particular attention to the questions of synchronization and flow control.

Finally, in Chapter 9, we present conclusions.

2. Objects

Most current programming languages are based on the notions of *variables* and *values*. Values are mathematical values, such as integers, characters, and sequences. Variables are cells that *contain* values. Assignment *copies* a value into a variable, destroying the previous contents of the variable.

A number of languages, such as LISP, Simula, and CLU, are based on the notion of *objects*. In this chapter, we describe this notion of objects and its extension to a very large universe of objects. We discuss the programming advantages and some of the implementation implications of the object-oriented storage model.

2.1 Description

The concept of objects is best explained by describing properties of objects. First of all, objects are the information-containing entities that are created and manipulated by programs. Thus, integers, strings, arrays, and procedures are all examples of objects.

Second, the information content of an object may include other objects. Although some objects (e.g., integers) are normally thought of as being unstructured, others (e.g., arrays) are normally thought of as having components that are themselves objects.

Third, a single object can be a component of many objects or be denoted by (be the "value" of) many variables. For example, in a given program, two variables x and y can both denote the integer 3.

Fourth, an object, once created, exists forever. There is no concept of the "extent" of an object. An object exists as long as the program needs it; it is up to the language implementation to determine when the storage for an object can safely be reclaimed.

So far, all of these properties hold for the values in common programming languages. Such values, being mathematical values, indeed exist forever conceptually. It is certainly true that an array value is composed of element values and that a single value can be the component of many structured values and can be the value of many variables.

What distinguishes objects from values is that objects can have a time-varying information content, or *state*. Objects with a time-varying information content are called *mutable* objects. Objects with a constant information content are called *constants*, and are equivalent to values in traditional programming languages.

For example, in CLU, integers, characters, and strings are all constants. CLU arrays, however, are mutable. The information content of an array includes some number of element objects. An array object can be modified by use of the *store* operation, which replaces one element of the array with a new object. For example, if *a* denotes an array that contains the elements 3, 4, and 7, then the effect of *store* (*a*, 1, 6) is to change *a* so that its first element is now 6 (*a* now contains the elements 6, 4, and 7).

In a (strict) value-oriented language, all mutability occurs as the result of assignment. The analog of the operation invocation *store* (*a*, 1, 6) is the assignment *a*[1] := 6. This assignment would be viewed either as assigning a new array value (equal to the old one except at index 1) to the array variable *a* or (in a more complex model) as assigning a new integer value to the integer variable *a*[1].

The addition of mutable objects gives new meaning to the properties listed above. Note that a mutable object has an identity above and beyond its current state. Two different arrays can both contain the same elements, but if a *store* operation is performed on one of them, the other one will not change. However, if a single array object is denoted by two variables *x* and *y*, then a modification to the array made via one variable will be visible via the other variable. The array is said to be *shared* by the variables *x* and *y*. Assignment in an object-oriented language causes a variable to denote the object resulting from the evaluation of the right hand side. An assignment of the form $x := y$, where *x* and *y* are variables, causes *x* and *y* to share the object originally denoted by *y*.

Mutable objects are not always structured, nor are all structured objects necessarily mutable. For example, one could define *counters* that are mutable objects similar to integers. One could also have (mathematical) sequences, which are immutable analogs of arrays.

2.2 Implementation Implications

The possibility of having shared, mutable objects has significant implementation implications. Without mutable objects, the concept of sharing is not particularly useful. If two variables *x* and *y* both denote a sequence value *v*, then one could say that the sequence value *v* is shared by *x* and *y*. However, because *v* is a constant, the implementation is free to have separate copies of *v*, one for each variable that denotes it. With side effects on *v* impossible, no one can tell if multiple copies are being used or not.

Thus we arrive at the traditional implementation technique for value-oriented languages. Each value is represented by a constant bit string. Each variable is represented as a container that can contain a bit string. Assigning a value to a variable is implemented by *copying* the bit string representing the value into the container representing the variable. There is no problem determining when to throw values away: when a variable is deleted (upon block exit), its contents are deleted.

In an implementation of an object-oriented language, there can be only one copy of each mutable object. A variable denoting some object will contain a *reference* to the object. A reference is a bit string that logically contains two items of information: One item is a *type code*; the type code indicates the type of the object to which the reference refers. (The type code would be *integer*, *string*, etc; in many implementations, a full type code is not required.) The second item is the *data part*; it in some manner identifies or names one particular object of the specified type. Typically, the data part is the address of a block of storage in which the state of the object is stored. Assignment copies a reference to an object, not the object itself.

The property that an object exists forever has a non-trivial meaning for mutable objects. There may be many references to an object; as long as there is at least one reference accessible to the program, the object must be retained. Thus, some form of automatic storage reclamation (e.g., garbage collection) is necessary to implement an object-oriented language.

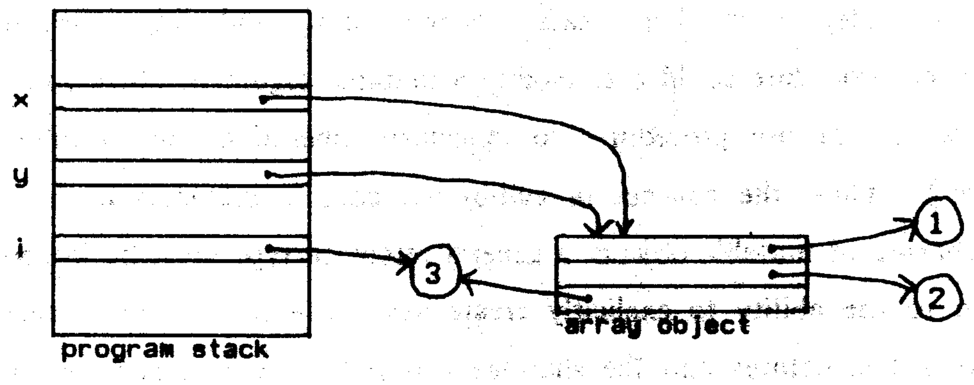
2.3 Example

Figure 1 shows an example of shared objects. *x*, *y*, and *i* are local variables of some process. Local variables are shown as cells allocated in a stack. (Although the stack looks like an object and may be implemented as an object, it is not directly accessible to the program.) Variables contain references to the objects they currently denote; in the figure, references are depicted as arrows. The variables *x* and *y* both denote the same array object, whose current state contains three elements, the integers 1, 2, and 3. The variable *i* also currently denotes the integer 3.

2.4 Advantages

Object-oriented languages have a number of advantages. The primary advantage is direct support for mutable objects. The use of mutable objects is common in programming. However, traditional value-oriented languages do not

Figure 1. An example of objects.



support mutable objects well. As we shall describe below, strict value-oriented languages have to be extended to provide the facilities needed to use mutable objects. The resulting languages are unnecessarily complex and error prone.

In addition, most value-oriented languages provide completely separate mechanisms for accessing "permanent" data stored in a file system. A file system is most naturally described as a shared, mutable data structure (the file directories) containing mutable or immutable objects (the data files). The file system is a permanent data structure; the creation and deletion of files do not correspond to any stack discipline. An object-oriented storage model allows the permanent file system and the temporary data of programs to be unified in a single universe of objects.

Object-oriented languages allow mutable data abstractions to be directly modeled. In an object-oriented language, mutable objects are "first-class citizens." They can be created, assigned to variables, passed to procedures, and returned by procedures. The allocation and deallocation of storage is completely handled by the language implementation.

In a value-oriented language, the closest thing to a mutable object is a variable. However, to use a variable as a mutable object, one must be able to share the variable, at the very least between calling and called procedures. (Otherwise, no procedure could ever modify a mutable argument object, in which case one could not use procedures to implement operations on mutable data abstractions.) Thus, the concept of call-by-reference is introduced. To allow arbitrary sharing of mutable objects, a general reference type must be introduced, in addition to the ability to explicitly create new variables. While a reference type allows list structures and the sharing of objects in data structures, it also introduces problems of dangling references. In most common programming languages, variables are deleted either implicitly upon block exit or explicitly by user command. In either case, it is possible that references to the deleted variable still exist.[1] If one attempts to use a dangling reference, the system should prevent the attempted use and report an error. However, in most systems such checking is deemed too expensive and is omitted, in which case use of a dangling reference is likely to cause havoc.

In an object-oriented language, dangling references are impossible, as objects are deleted only by the system when they are no longer accessible. In addition, the concept of a reference is implicit in the semantics of an object-oriented language. No explicit reference type is needed, nor is call-by-reference. When a procedure is invoked, it is passed a number of objects as arguments. These objects are *assigned* to the formal arguments of the procedure; they are thus *shared* between the calling and called procedures. If an argument object is mutable, then it may be modified by the called procedure; if it is constant, then of course it can never be modified. Because there is no reference type, there is no possibility of references to local variables; thus, the variables of one procedure can be completely isolated from access by any other procedure. In an object-oriented language, variables can simply be local names used within a single procedure to refer to its objects.

As we stated earlier, in the implementation of an object-oriented language, mutable objects are accessed via references. While it is not necessary to do so, it is quite convenient to access all objects by fixed-size references.[2] (We do not mean to imply that integers must be allocated in a free-storage area. Because

[1]In Algol 68 [34], dangling references to implicitly-deleted "loc" variables are prevented by scope rules that forbid a reference to a variable of "newer" scope from being assigned to a variable of "older" scope (the "newer" the scope, the sooner the variable will be deleted). Such scope rules tend to be excessively restrictive. For example, these scope rules make it impossible for a procedure to create a new object and insert it into a previously-existing structure passed to it as an argument. To overcome these limitations, Algol 68 also provides "heap" variables, which are similar to objects (except that references are explicit).

integers are constants, the data part of an integer reference can be the integer value itself. This technique can be used for all constants whose information content will fit in a reference.) If all objects are accessed via fixed-size references, then each variable is simply a cell that can contain a reference. Because all variables are the same size, the "size" of an object becomes much less important than in a value-oriented language where the "size" of a value determines the size of the variable that must hold it. In particular, a compiler for an object-oriented language does not have to be concerned with the sizes of objects when generating code.

One result of using fixed-size references is that it is easy to provide objects (e.g., arrays) that can grow and shrink dynamically to whatever size is needed. Use of such dynamic objects eliminates unnecessary size limits in programs and also probably saves space: instead of allocating a maximum size (which is selected in the hope that it will never be exceeded), only the amount of storage actually needed is allocated. Similarly, it is possible to efficiently implement "unbounded" integers in an object-oriented language. Small integers can be represented by references that contain the integer value. Larger integers can be represented by references that point to separately-allocated storage. The use of unbounded integers would remove a major source of machine dependency that exists in most programming languages.

In addition, use of fixed-size references for all objects facilitates separate compilation of modules. In a language where a module may define a data type, modules that use the data type can be compiled before a representation has been chosen for the data type. The compiler can generate code that uses objects of that data type without having to know how "big" the objects are. Similarly, the

[2]This implementation technique can of course be used for value-oriented languages as well as object-oriented languages, but it usually isn't.

implementation of parameterized modules is facilitated. An example of a parameterized module is *stack*[*t*], where *t* is a type parameter that specifies the type of the objects in the stack. A compiler can generate a single (possibly parameterized) object code module for stacks that will work regardless of what type of element is actually being used. Finally, because all variables are alike, it is possible to have a type *any*. A variable of type *any* is allowed to refer to any type of object, thus providing an escape mechanism to run-time type checking. Such a notion is impossible in a traditional implementation of a value-oriented language, where the size of variables is not bounded.

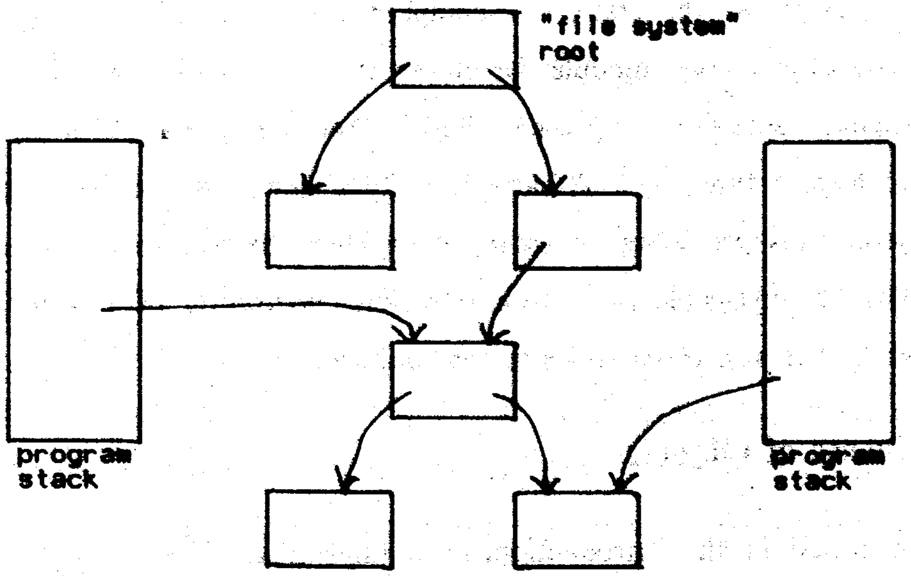
2.5 A Universe of Objects

As we stated in the Introduction, most implementations of object-oriented languages provide a single space of objects for each program or process. Each space of objects is usually small, and no communication of objects between different spaces is possible. We are proposing to extend the notion of objects by providing a single, very large space of objects that would be shared by all processes in the system. This universe of objects would include not only the temporary objects used during the execution of programs, but would also include "permanent" objects normally stored in a file system.

Figure 2 shows an example of a (small) universe of objects. In this example, there are two process stacks representing two running processes. In addition, there is a small file system attached to a root node. Note that some objects are both part of the file system and referred to by processes.

Having a single system-wide universe of objects has a number of advantages compared to current systems. The first advantage is uniformity. All of the data in the system, both local data used by programs and permanent data, are objects. All such data are treated the same (except that objects that may be shared by

Figure 2. A universe of objects.



multiple processes may require synchronization). In current systems, there are two kinds of data: variables and files. The two kinds of data have vastly different characteristics.

The second advantage is simplicity. Permanent data objects are operated upon directly, just like local data objects. No explicit I/O need be performed by the programmer; no conversions are needed. In many current systems, files can be accessed only by performing explicit I/O to or from local variables. In addition, objects must be converted between their in-memory format and an external format (either a string of bits or a string of characters).

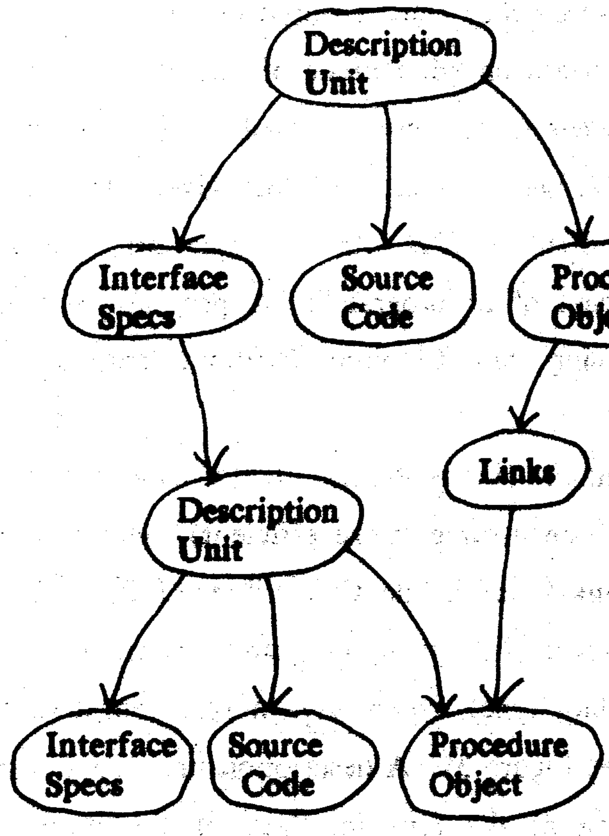
The third advantage is generality. There are no restrictions on what kinds of objects can be permanently stored. For example, one may store large numbers of small objects or objects that involve list or graph structures. Current systems generally do not efficiently support small files (files whose sizes are comparable to programming language objects, i.e. 1 to 100 words). In addition, few systems permit pointers to be stored in files. Data structures that involve pointers must be converted to some other form before they can be stored in a file.

Similarly, there are no restrictions on what kinds of objects can be passed between procedures, between processes, or between programs. Conventional systems generally limit the "arguments" that can be given to programs (subsystems) to constant values, usually strings. In an object-oriented system, there may be no need for a concept of a "program". A program is simply a procedure; normal argument passing can be used (without restriction) for communication.

The net effect of these advantages is that the use of long-lived data bases is encouraged and made easier (and, we assume, more efficient). An example of the kind of data base we are thinking of is the CLU library [22]. The CLU library is a data base containing information about programs. Each module of a program (e.g., a procedure or a data type) would have a representative in the library called a *description unit* (see Figure 3). A description unit would contain, among other things, a specification of the type interface of the module. A procedure description unit would contain the number and types of the procedure arguments; a data type description unit would contain interface specifications of the operations of the data type. Note that the interface specification of a procedure contains data types and thus may refer to data type description units. The interface specifications would be used by the CLU compiler to type-check intermodule references (e.g., calls of one procedure by another).

The CLU library can naturally be described as a set of objects with inter-object references. Unfortunately, conventional systems do not directly support such a notion. Implementing the CLU library on a conventional machine is difficult, particularly because of the relatively large numbers of small objects

Figure 3. The CLU library.



involved. Furthermore, using such an implementation is difficult because the information stored in the library must be converted from the external file form into the internal object format used by the compiler. In a system supporting a universe of objects, the CLU library could be implemented directly as a collection of objects. These objects could be accessed directly by the compiler, without conversions. For example, the type descriptions used in the library could be exactly the same as the type descriptions used internally by the compiler.

3. An Object-Oriented Machine Architecture

In this chapter, we describe the visible architecture of the machine (the "machine language") and the philosophy of its design. Our purpose is not to fully specify the architecture, but merely to provide a context for later chapters, which describe the implementation of the machine. Many of the decisions made here represent personal preference and are not essential to the major ideas presented in the rest of the thesis.

3.1 CLU

The primary influence on the design of the machine architecture has been the programming language CLU [22]. CLU is an object-oriented language that has been designed to facilitate the construction of programs that are understandable, reliable, and maintainable. This goal is accomplished primarily by providing language constructs that support the use of abstraction in program design and implementation [21]. Three forms of abstraction are supported: procedural abstraction (procedures), control abstraction (iterators), and data abstraction.

Of these, the most interesting form is data abstraction, the definition and use of abstract data types. A data type in CLU consists of a set of *objects* and *operations*. The operations completely characterize the behavior of the objects: they are the only direct means of creating objects, obtaining information from objects, or modifying objects. CLU provides a number of primitive types, such as integers, booleans, characters, strings, arrays, and records. In addition, CLU allows the programmer to define new data types.

A new data type is defined by writing a module called a *cluster*. In a cluster, the programmer specifies a representation for objects of the new abstract type by giving a representation type. For example, a type *set*[*t*] may have a representation of *array*[*t*] (*t* is a type parameter specifying the type of elements in the sets), meaning that each object of type *set*[*t*] is actually represented by an object of type *array*[*t*]. In addition, implementations of the operations of the abstract type are given in the form of procedures that operate upon objects of the representation type. These procedures (only) are given the power to convert objects of the representation type into objects of the abstract type and vice versa. Thus only the operations of the type may directly create objects of the type.

To ensure that the behavior of an object of an abstract type is completely defined by the operations of the type, one must not allow the object to be operated upon as an object of the representation type. This restriction can be enforced by checking each procedure invocation to make sure that the actual argument objects are of the types expected by the procedure.

CLU has been designed so that complete type-checking can be performed at compile-time. Each variable in a CLU program is declared as to the type of object it may denote; each CLU procedure is declared as to the number and types of objects it accepts as arguments and returns as results. Since CLU expressions are composed of variable references and procedure invocations, the CLU compiler can determine at compile-time the types of all expressions and can check that all assignments and invocations are type-correct.

CLU also provides a type *any*, which allows a variable, procedure argument, or return value to be declared to be of any type. An expression of type *any* can potentially evaluate to any type of object. CLU has been designed so that in this situation explicit run-time type discrimination must be performed before the object can actually be used.

3.2 Architectural Philosophy

Our intention is to design a machine that will effectively support the implementation of a language similar to CLU. However, we are not proposing that the machine directly execute CLU programs. Instead, the machine will interpret programs in some intermediate language produced by a compiler. There are a number of reasons for making this choice. Interpreting an intermediate language is simpler and more efficient. Furthermore, as described above, a compiler can play an important role in the early detection of errors. (Because this "intermediate" language is directly implemented by the machine, it shall henceforth be called the *machine language*.)

The next issue is to determine the relationship between the machine language and CLU. One possibility is for the machine language to be simply a parsed form of CLU, with essentially the same semantics. However, we have decided that the machine language should not be semantically equivalent to CLU, but instead be at a lower level. The primary reason for this decision is to take advantage of CLU's ability to support complete compile-time type checking. Except where the type *any* is used, it is not necessary for abstract types to exist at run-time. Thus, it is not necessary that the machine directly support user-defined data types. As a result, the machine can be simpler and probably more efficient.

Although the machine language is not intended for direct use by programmers, we believe it should provide a simple and well-defined interface. Although it need not support user-defined data types, it should provide a fixed set of types, and all primitive operations should perform complete (run-time) type checking of their arguments. Similarly, where relevant, all primitive operations should perform bounds checking. All possible results of program execution should be defined in terms of the machine language; there should be no result that can be explained only by referring to details of the underlying implementation. The machine thus forms an opaque level that serves as a base for higher (software) levels that would implement such things as abstract data types. A system constructed in this manner as a hierarchy of opaque levels is likely to be both more understandable and more robust; faults resulting from compiler errors or hardware malfunctions will tend to be caught earlier and by higher levels of the system.

The machine language will be designed to support CLU; we are not trying to design a machine language that would be generally useful for implementing any programming language. Thus, the machine language types should be useful for implementing the primitive CLU types. However, they need not be exactly the same as the CLU types. For example, the CLU *array* type is parameterized: each array may hold objects of only one type. The corresponding type in the machine language need not have this restriction.

Certain decisions about the machine language must be made that go beyond the current design of CLU. For example, we must specify some mechanism for the creation of multiple processes and the sharing of objects among multiple processes. Other unresolved issues, such as inter-module linking, multiple implementations of types, and protection, will be ignored.

3.3 The Machine Language

The machine language provides a fixed set of data types, each with a fixed set of primitive operations. Of these data types, two are of particular importance, *procedures* and *processes*.

A *procedure* is an object that can be executed by the machine. A procedure accepts a fixed number of objects as arguments and produces a fixed number of objects as results. The machine provides some number of primitive procedure objects, most corresponding to operations of the primitive types. In addition, the machine provides a way in which new procedure objects can be created. These user defined procedures correspond to machine language programs. When invoked, such procedures are *interpreted* by the machine. The basic actions that can be performed by a machine language procedure are defining and assigning to local variables, invoking procedures, and performing other control functions such as conditionals and looping. A procedure is quite limited in its ability to access objects. A procedure can access only its arguments, its local variables, and a fixed set of known objects specified at the time the procedure was created. These known objects would include the procedure objects that are to be invoked by the procedure. (A possible machine language is presented in Appendix I.)

A *process* is an object that represents the potentially concurrent execution of a procedure. Process creation is similar to procedure invocation. A procedure object and a set of argument objects are specified; the procedure is invoked with the specified arguments. (As in the case of normal procedure invocation, the argument objects are shared by the caller and the called procedure.) However, instead of returning the results of the invoked procedure, the process creation operation immediately returns to its caller a process *object*. The execution of the invoked procedure will proceed concurrently with the execution of the caller. Process operations can be performed on the process object; operations are provided to start or stop the execution of the invoked procedure and to determine whether the procedure has terminated and, if so, what the result objects are (see Figure 4).

One run-time evaluation stack per process is used by the machine to store procedure arguments and local variables. However, these stacks are implicit in the semantics of the machine language and are not directly accessible to procedures. We assume that when the machine is initialized, a single process is automatically created to execute an appropriate start-up procedure. We do not specify how an initial universe of objects is created.

Figure 4. Process operations.

```
process_state = oneof [
          stopped: null,
          killed: null,
          terminated: array[any], % the result objects of the procedure
          running: null  % none of the above
          ]

create = proc (p: procedure, args: array[any]) returns (process)
          % The new process is created in the stopped state.
start = proc (p: process)
          signals (process_terminated) % If process terminated or killed.
stop = proc (p: process)
kill = proc (p: process)
state = proc (p: process) returns (process_state)

block = proc ()
          % Blocks the executing process until a wakeup is performed.
wakeup = proc (p: process)
          % Wakes up the specified process, if blocked.
```

The other machine-language types are ordinary data types that correspond in some manner to the primitive CLU data types. Thus, we would expect the machine to support integers, booleans, characters, and strings, plus some structured objects. A number of types of structured objects could be provided, such as fixed-length structures (corresponding to CLU *records*) and variable-length structures (corresponding to CLU *arrays*). The exact choice of types is not particularly important for the purposes of this thesis.

The set of accessible objects includes all running (or runnable) processes. In addition, we assume there is a single distinguished object, called the *root* of the file system, that is always accessible. This object is the root of a tree-like (or graph-like) directory structure that contains the "permanent" objects of the system, e.g., programs and data bases. Naturally, any object that is referred to by an accessible object is itself accessible. All other objects are by definition inaccessible and their storage subject to being reclaimed by the system.

We will define all primitive operations to be atomic. By atomic, we mean that any set of primitive operations performed concurrently must be equivalent to performing the same set of operations in some order. This definition follows from our desire that the behavior of machine language programs be well-defined. Its implication is that the machine must synchronize operations on mutable objects to ensure consistency.

Of course, providing synchronization for the primitive types will not eliminate the need for explicit synchronization of user-defined objects. As yet, CLU provides no synchronization mechanism (it is a sequential language). A number of synchronization mechanisms have been proposed in the literature, such as semaphores, monitors, eventcounts, and serializers. However, the search for the "best" synchronization mechanism is a subject of current research. Therefore, rather than choosing one of these mechanisms, we will provide the most basic synchronization primitives, *block* and *wakeup* [28] (see Figure 4). Together with shared objects, these primitives can implement any of the proposed mechanisms.

3.4 A Simplification

Although we believe that the machine language should provide data types that are similar to the primitive CLU types, the implementation of such types would involve a fair amount of detail that would serve no useful purpose in this thesis. Thus, for presentation purposes we will assume that the machine provides (in addition to procedures and processes) only two data types, *bstrings* and *vectors*.

Bstrings are constant (immutable) fixed-length bit strings. The length of bstrings will be chosen so that a bstring value can be stored entirely in the data part of a reference; thus, no additional storage is needed to implement bstrings. (The subject of reference size is discussed in Chapter 5.) Bstrings are essentially equivalent to the untyped data manipulated by conventional machines. The operations on bstrings would be the usual arithmetic and logical operations; the exact choice of operations is not particularly important for our purposes. However, there would be no *create* operation. New bstring values are created by performing operations on old values. We assume that bstrings are ultimately created by I/O devices. Naturally, bstrings can be interpreted as characters, booleans, or small integers as the need arises.

Vectors are fixed-length, mutable collections of objects, similar to arrays in most programming languages. We assume that the elements are numbered starting from zero. We will place an upper bound on the maximum size of a vector. The exact choice is a matter for the system designer and is dependent upon a number of factors (see Chapter 7). The maximum vector size will likely be in the range from 128 elements to 4K elements.

The vector operations are listed in Figure 5. The *create* operation creates a vector with a given number of elements, all of which are initialized to some distinguished bstring object. (In a real system supporting many primitive types, this object would be the unique *undefined* object, whose purpose is to permit detection of use of uninitialized variables and vector elements.) The *size* argument is a bstring that is interpreted as an integer. If the *size* argument is less than zero or greater than the maximum size of a vector, then an exception is signalled. (We assume that the machine language supports some form of exception handling similar to that in CLU [23].) Similarly, an exception is signalled if there is insufficient storage available to satisfy the request. In all cases we assume that an exception is signalled if an object of the wrong type is given to a primitive operation.

Figure 5. The vector operations.

```
create = proc (size: bstring)
                 returns (vector)
                 signals (negative_size, size_too_large, no_storage)
equal = proc (v1, v2: vector) returns (bstring)
size = proc (v: vector) returns (bstring)
fetch = proc (v: vector, index: bstring) returns (any) signals (bounds)
store = proc (v: vector, index: bstring, element: any) signals (bounds)
```

The *equal* operation returns true (a particular bstring) if its two vector arguments are the same vector object, and returns false (a different bstring) otherwise. (Two vectors are not equal just because they currently have the same contents; each invocation of the *create* operation returns a vector that is distinct from any previously created vector.)

The *size* operation returns a bstring (interpreted as an integer) that specifies the number of elements in the vector. The *fetch* operation returns an element of a vector given its index. The *store* operation modifies the vector to contain the given element. Both *fetch* and *store* signal *bounds* if the index is less than zero or greater than or equal to the size of the vector.

These types are quite primitive and would seem to contradict many of the claimed advantages of object-oriented languages given in Chapter 2. However, we are not proposing that a real machine provide only these types, or that it would provide these types at all. We have chosen these types to simplify the presentation in the remainder of the thesis.

The bstring and vector types are realistic in that they could be used internally to a machine to construct more useful types. For example, larger structured objects could be constructed by using two levels of vectors: the object would be represented by a single top-level vector containing references to lower-level vectors that store the actual object elements. This usage would be equivalent to the use of page maps in current paging systems. Dynamic structured objects (objects that can grow or shrink) can again be implemented by using two levels of vectors. Adding or deleting storage can be performed by adding, deleting, or replacing the lower-level vectors. This implementation is equivalent to our current implementation of CLU arrays on a conventional machine. Unbounded integers could be implemented by using bstrings for small integers and vectors of bstrings for larger ones.

4. System Structure: an Overview

In this chapter we provide an overview of the design of a machine to implement the language described in the previous chapter. We begin by reviewing the functions that the machine must perform.

4.1 Machine Functions

The primary functions of the machine are to implement the four machine language types: processes, procedures, vectors, and bstrings. The implementation of processes involves the management of a collection of concurrent activities. The machine must assign resources to those activities in some reasonable way. The machine will need to maintain state information for each process. Naturally, the various process operations, including *block* and *wakeup*, must be implemented.

The implementation of procedures involves primarily the interpretation of machine language code. This interpretation will involve the manipulation of an evaluation stack and the invocation of the primitive operations.

The implementation of vectors involves the allocation, management, and automatic reclamation of storage. The amount of storage provided is sufficiently large that a multi-level memory system is required. Implementing the vector operations also requires that the machine be able to map from a vector reference to the actual storage for the vector. In addition, the various vector operations must be synchronized so that they behave as atomic operations.

Vectors are the basic storage type of the machine. They can be used to store information needed for the implementation of processes and procedures. A procedure can be represented by a vector containing instructions (encoded as bstrings) and references to "known" objects (literals and other procedures). A process can be represented by a state vector, which contains such information as the currently active procedure, the instruction counter (which identifies an instruction in the active procedure), the evaluation stack, plus scheduling information. A procedure or process reference would thus actually be a reference to the corresponding vector representing the procedure or process. These uses of vectors would not be apparent at the machine language level.

The implementation of bstrings involves simply implementing the bstring operations. No additional "storage" is needed, since (by definition, see Section 3.4) the value of a bstring can be completely contained in its reference.[1] The implementation of bstrings is thus trivial, and can be performed directly by any hardware module. For example, the invocation of bstring operations by machine-language procedures can be performed directly by the machine-language interpreter.

There are some other functions that the machine must perform that are not directly related to the implementation of any particular data type. For example, the machine must perform system initialization, crash recovery, and reconfiguration. In addition, the system must provide some form of I/O. The implementation of these functions is not discussed in this thesis.

4.2 Design Strategies

In this section we describe some of the design strategies used to make the machine as simple and understandable as possible. The primary technique used to minimize the complexity of the machine is modularity: splitting the machine into separate modules with well-defined interfaces. A modular design is easier to understand because it is composed of a number of parts of a more manageable

[1]The actual storage is provided by vectors (whose elements may be bstrings) and by hardware registers.

size, each of which can be examined and understood separately. In a good modular decomposition of a system, each module can be viewed as a "black box" at the system level. That is, when viewing the system as a whole, one need understand only the interface of each module, and not the internal construction of the module. Similarly, when viewing the internal construction of a module, one need only relate it to its interface specification; the other modules in the system can be ignored. This approach can be applied hierarchically, as any module can itself be constructed out of a number of internal modules.

There are certain strategies that can be used to obtain a modular decomposition of a system. One strategy is separation of function: providing a separate module for each function that must be performed by the system. This idea is related to our desire to minimize the use of multiplexing in the system. In conventional systems, multiplexing often involves the use of a single module to perform many functions. For example, a single processor is multiplexed to interpret user programs, implement the virtual memory, and control I/O devices.

A useful technique for identifying functions that can be implemented by modules is the notion of data types. A module that implements a data type encapsulates knowledge about the implementation of the objects of the type. Other modules can use the objects without knowing any details of their implementation. The objects are identified by references; however, the references are interpreted only by the type module, which performs operations upon the objects at the request of other modules.

As we described above, many of the functions of the machine correspond to the implementation of particular data types (processes, procedures, vectors, and bstrings). Except for bstrings, each of these types is a reasonable candidate for implementation by a separate module. It may be convenient in the implementation of these types to introduce subsidiary types; for example, in the implementation of vectors, a type *page* may be useful.

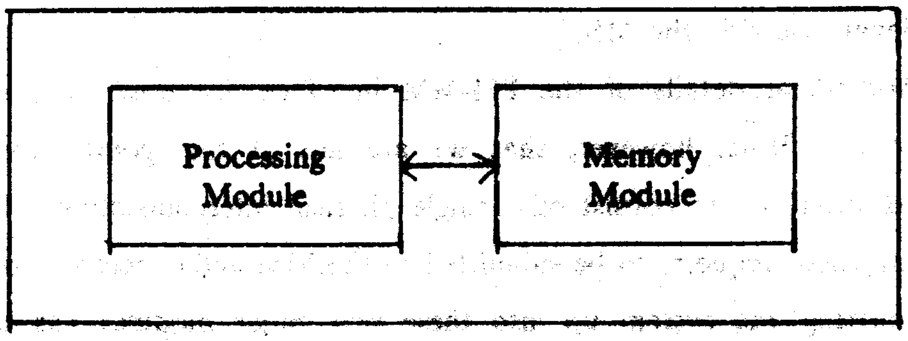
An important problem area in the design of most computer systems is the proper synchronization of concurrent activities. The system should be designed so that there are no undesirable race conditions and no possibility of unintentional deadlock. The methods used to implement synchronization should be both efficient and easily understood. The use of data types can be helpful here. Because all operations on an object are actually performed by a single module (the type module), that module is in an excellent position to supervise the concurrent execution of operations to ensure consistency. We will discuss this issue further in later chapters.

One useful design goal is that all module interfaces be speed-independent. Speed independence means that the system will work regardless of the time taken to transmit data from one module to another or the time taken for a module to respond to an input. Speed-independence is a method of avoiding race conditions. If some action happens to take a long time, the performance of the machine may be degraded, but the machine will still function "correctly".

4.3 High-Level System Structure

We are now ready to describe the overall structure of the system. Our first decision was to separate the implementation of procedures (instruction interpretation) from the implementation of vectors. We split the system into two major parts, a *processing module* (PM), which interprets procedures and supports multiple processes, and a *memory module* (MM), which implements vectors (see Figure 6).

Figure 6. High-level system structure.



The interpretation of procedures involves performing explicitly invoked vector operations. In addition, as described above, both procedures and processes are actually represented by vector objects. Thus, the PM uses the MM to assist in the implementation of procedures and processes.

The communication between these two modules consists primarily of requests sent from the PM to the MM and replies sent from the MM to the PM. These requests correspond to the primitive vector operations, *create*, *equal*, *size*, *fetch*, and *store*. Each request message consists of a fixed amount of information, which includes an identification of the operation to be performed, plus object references for the arguments of the operation. Reply messages contain the status of the reply (normal or exceptional termination), plus object references for any results of the operation.

For example, suppose a procedure invokes the vector *create* operation to create a five-element vector. When this invocation is executed by the PM, it will send a *create* request to the MM. The MM will create a new vector, initialize it, and return a reference to the vector to the PM, which will use that reference as the result of the invocation. The request and reply messages are shown in Figure

7. (The notation *t#d* indicates an object reference with type code *t* and data *d*. In the case of a vector reference, the exact data value is not predictable, nor is it relevant outside the MM.)

Additional details of the PM-MM interface are presented below and in Chapter 8. Note, however, that we are not at this point constraining the PM-MM interface to consist of a single physical interconnection. The interface allows multiple requests to be submitted to the MM and processed concurrently.

Splitting the system up into these two major modules has a number of advantages. The primary advantage is that the structure of the system is simplified, compared to conventional systems. Although the module diagram above may look like a conventional system, there is a crucial difference. A primary memory in a conventional system is a low-level hardware unit that is but one piece of the implementation of the virtual address space seen by user programs. While a conventional primary memory has a well-defined task and a simple interface, the relevant function (the user-visible virtual memory) is not implemented by a single module, but by a collection of hardware modules and system processes.

The MM, on the other hand, directly and fully supports objects that are very similar to those manipulated by user programs. The MM encapsulates all knowledge of how vectors are implemented, including the implementation of the

---

Figure 7. An example request and reply.

Request:	**Reply:**
name: *create*	status: normal
size: bstring#5	result: vector#?

multi-level memory, storage allocation, and automatic storage reclamation. The MM assumes full responsibility for implementing vectors. Any processing power needed to perform this function will be provided within the MM.

The sole function of the PM is to implement procedures and processes. Unlike conventional systems, the PM executes no "privileged" machine-language code to support the multi-level memory system. The interface between the PM and the MM is clean and high-level, consisting basically of invocations of the vector operations. There is essentially only one "rule" that the PM must obey: it must not create or modify vector references. If this rule is obeyed, then there is no way that the PM (and therefore, user programs) can interfere with the correct operation of the MM.

Actually, there is one additional interaction between the PM and the MM. The PM must cooperate with the MM to allow the MM to determine which objects are needed and which can be reclaimed. In particular, at certain times the MM will request the PM to discard all of its vector references (except references to the root vector, which is permanently accessible). One way for the PM to satisfy this request is to store its references in the MM, in vectors accessible from the root. When there are no vector references outside the MM, then we say the system is in *quiescence*. During quiescence, the MM can examine the entire collection of accessible vectors, without interference from the PM. (Exactly what the MM does is the subject of Chapter 6.) The MM informs the PM when it is finished. The PM will then read back all needed data from the MM and resume normal operation.

This additional interaction between the PM and the MM is not desirable, but is probably the best alternative. The definition of quiescence is easy to understand, and verifying its correct implementation should be straightforward. Cooperation between the PM and the MM is needed because the implementation of automatic storage reclamation requires that all vector references be accounted for, regardless of where they are in the system. The need for cooperation can be eliminated only if vector references never leave the MM, as in an architecture proposed by Baker [2]. In such a solution, some means must be provided to allow the PM to identify particular vectors in its requests to the MM; in Baker's proposal, the PM would specify one of a number of special registers in the MM. The effect is to introduce a new "address space" (e.g., register numbers) used instead of references outside the MM. These "addresses" are inferior to references for a number of reasons. For one thing, the relationship between addresses and objects is time-varying; an address is valid only as long as the register is unchanged. In addition, it is impossible to name any existing object at any time (the problem of register allocation is introduced). For these reasons, we consider such solutions less desirable.

4.4 Multiprocessing Philosophy

Our system uses multiple processors to implement multiple processes. This section describes our motivation for using multiple processors and explains the reasoning behind a number of related decisions: (1) to not switch processes on secondary storage accesses (page faults), (2) to store process state vectors in the virtual memory, and (3) to prohibit the preemption of a process while waiting for a reply from the MM.

Our goal in using multiple processors is not so much to increase the capacity of the system, but to reduce the need for processor multiplexing. Current systems attempt to maximize the utilization of the (usually single) processor through the use of *short-term scheduling* [27]. Short-term scheduling is a technique by which a processor is multiplexed among a small number of processes (called the *eligible* processes). Whenever the currently executing process must wait (e.g., for disk I/O), another eligible process is quickly selected for execution. For maximum processor utilization, the time required to switch the processor from one eligible process to another should be small. The number of eligible processes is selected to satisfy two constraints: (1) There should be enough eligible processes that at least one is always ready to run. (2) There should be sufficiently few eligible processes that their working sets can all be contained in primary storage. Short-term scheduling is distinct from long-term scheduling, where the primary goal is the fair distribution of resources to processes (possibly of differing priorities). Long-term scheduling operates by determining the set of eligible processes; this set changes at relatively long intervals (on the order of 100 milliseconds).

If processors are relatively inexpensive, then multiple processors can be used, and processor utilization becomes less important. Instead of using short-term scheduling to multiplex a single processor among a set of eligible processes, we can run each eligible process on a separate processor. Process switching will still be necessary to support long-term scheduling; we would always expect the number of active processes to be larger than the number of processors. However, the rate of process switching will likely be less, so that the time required to switch processes will be less important.

The performance emphasis in this design is not the processor utilization, but the execution speed of individual processes. Improved execution speed is obtained by providing additional processors and assigning each process to a processor for longer periods of time. During that time, the process will obtain greater use of its processor, as the processor is not being shared with other processes. In addition, the longer a single process occupies a processor, the more effective use it can make of a local cache in the processor.

Using multiple processors increases the demand on the MM. If the MM bandwidth is inadequate, processes will be delayed because of memory contention. Adequate MM bandwidth can be provided using a technique analogous to conventional interleaving, described in Section 8.6.

Once we abandon short-term scheduling, it no longer makes much sense to switch processes when a process accesses secondary storage (takes a page fault[2]). Conventional systems preempt a process when it takes a page fault so that another process can be run while the first is waiting for the page to be brought into primary storage. Because we are assuming secondary storage devices that are significantly faster than current secondary storage devices, the delay caused by accessing secondary storage will be much shorter in our system than in current systems. Thus, switching processes on page faults would not result in much improvement in processor utilization. Furthermore, as described above, we do not demand that processor utilization be maximized. Therefore, there is no need to switch processes on page faults.

The decision to switch processes on page faults in conventional systems requires that all information needed to perform a process switch be available in primary storage. If process switching could produce a page fault, then processor utilization could be degraded. More importantly, the system would have to be designed to handle page faults in the page fault handler. Switching a processor between processes involves writing information into the old process state vector and reading information from the new process state vector. Current systems

[2]We use the term page fault to indicate the situation where a request to the MM requires accessing secondary storage. If we wanted to switch processes on secondary storage accesses, we would have the MM notify the PM when a request initiates a secondary storage access. The PM would then have the opportunity to begin executing another process. Unlike conventional systems, however, the PM would not have to do anything in response to a page fault. The page fault notification would simply be advice designed to allow improved performance. Regardless of that the PM did, the MM would perform the secondary storage access and complete the requested operation.

avoid these problems by ensuring that the state vectors of all processes (or all *eligible* processes) are "wired-down" in primary storage, so that accessing a state vector can never generate a page fault.

In our system, we do not switch processes on page faults. It is thus possible to store process state information in ordinary objects in the MM. This decision avoids the need for a special mechanism to provide wired-down storage for process state vectors. In addition, it avoids the need to place a limit on the number of existing processes. We would like to encourage the use of processes wherever natural. In particular, we would like to encourage the use of large numbers of long-lived processes that spend most of the time waiting for some event to occur. Although process switching may occasionally be delayed because of secondary storage accesses, the average process switching time should not be degraded, as active process state vectors would naturally tend to remain in primary storage.

In summary, we have decided (1) not to switch processes on page faults and (2) to store process state vectors in virtual memory (no "wired-down" vectors). Given these decisions, it is reasonable to go one step further and disallow preemption of a process while it is waiting for a reply from the MM. (Any preemption would be delayed until the MM has replied.) In effect, all requests to the MM are like uninterruptible procedure calls. This decision leads to a significantly simpler system structure. As far as the PM is concerned, there is no such thing as a page fault. Some requests to the MM are answered quickly, and some take more time. Interactions between page faults and process management are a major source of complexity in current systems. In our system, there are no such interactions.[3]

The disadvantage of prohibiting preemption of processes waiting for replies from the MM is that the worst case preemption time becomes longer.[4] To avoid the need for fast preemption, we will assume that all I/O devices are interfaced via separate controllers that isolate any severe timing constraints from the rest of the system through the use of buffering.

The area of interruptibility is a problem in most multiprogrammed systems. When a process is preempted, its state must be saved so that it can later be resumed. The best time to preempt a process is when it is "between instructions," that is, when there are no activities in progress in the processor and the process state is well-defined. If a process is preempted while instruction execution is in progress, then additional state information may need to be saved to record the progress of the interrupted instruction. In addition, if the instruction involves obtaining exclusive access to some shared object, then the process should not be preempted until the shared object is released. Otherwise, the preempting (higher priority) process may hang waiting for the object to be released, possibly causing deadlock. This latter problem can occur in current systems when a process executes a "supervisor call" instruction.

If rapid interrupt response is not necessary, then allowing preemption only "between instructions" is acceptable, provided the instruction execution time has a reasonable upper bound. In our case, we must ensure that all primitive procedures (those implemented directly by the machine) will terminate in a reasonably short time. Thus, for example, we have defined the vector *create*

[3]A similar position has been taken by the designers of the M.I.T. LISP machine [4]. In that machine, the virtual memory fetch and store operations are uninterruptible microcode routines. This decision was based on a desire to simplify the implementation and encourage the use of the virtual memory by all parts of the system. (Routines that cause interrupts are more difficult to use, especially by routines that handle interrupts.)

[4]The worst case involves transferring large (e.g., 4K-word) pages into primary storage. At a transfer rate of two words per microsecond, the page fault service time could be as high as two milliseconds.

operation to raise an exception if insufficient storage is available, rather than waiting until the request can be satisfied. As the availability of additional storage may depend upon the actions of other processes, there is no guarantee that additional storage will become available within any fixed time. If desired, a machine language procedure can be written that calls *create* repeatedly (at suitable intervals) until storage becomes available.

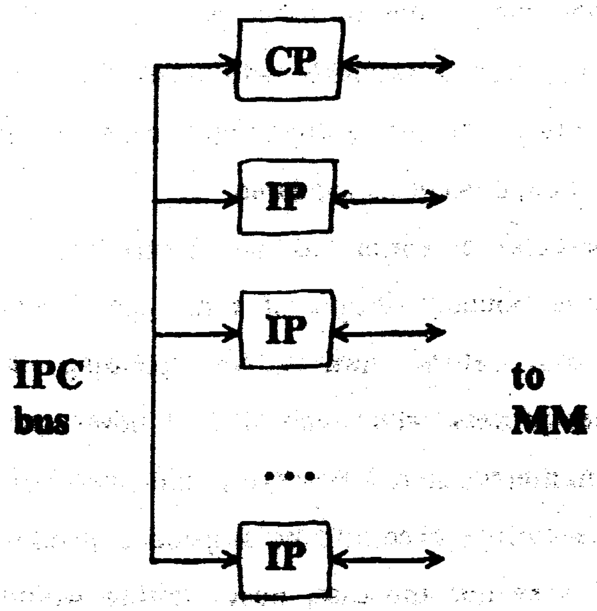
There is one essential exception to our restriction that all primitive procedures terminate in a bounded length of time: the *block* primitive. The *block* procedure does not return until a corresponding *wakeup* has been performed by some other process, which may in fact never occur. Thus, *block* could not be made uninterruptible. However, the motivation for *block* is explicitly to cause the executing process to be suspended pending the occurrence of the *wakeup*; if this were not the case, busy waiting would be satisfactory. Therefore, *block* is necessarily handled as a special case.

4.5 The Processing Module

The PM consists of one *control processor* (CP) plus some number of *instruction processors* (IPs), connected by an interprocessor communication (IPC) bus (see Figure 8). The function of an IP is to interpret procedures. An IP performs the computation of a single process (at a time); that process is said to be *bound* to the IP (and vice versa). At any time, each IP may be bound to at most one process, and each process may be bound to at most one IP.

The function of the CP is to manage the execution of multiple processes. The CP performs scheduling and controls the binding of processes to IPs accordingly. To perform scheduling, the CP maintains some database (e.g., a priority queue) that contains references to the (state vectors of) processes that are unbound and runnable (not terminated, blocked, stopped, or killed). Because

Figure 8. Processing Module block diagram.



unrunnable processes are not "held onto" by the CP, they are subject to being reclaimed should they become inaccessible. A stopped or blocked process can be made runnable only by performing an operation on the process object, which can happen only if the process object is accessible.

The CP and the IPs are each connected to a separate port of the MM. Each port accepts request messages one at a time; a processor must wait for a reply before sending another request. Requests submitted to different ports are processed concurrently by the MM; the order of arrival of requests on *different* ports is irrelevant.

An IP sends requests to the MM to perform vector operations invoked by the process it is executing. It also accesses the MM to fetch instructions and operands. The CP accesses the MM to manipulate process state vectors and its scheduling data base.

Process creation is implemented by sending a *create* message to the CP over the IPC bus. Contained in the *create* message is a reference to a vector that contains all information needed to initialize the new process. Upon receipt of a *create* message, the CP will create a process state vector, initialize it, and return the process reference to the requesting IP.

When the CP binds a process to an IP, it passes the process reference to the IP in a *bind* message. While the process remains bound to the IP, only the IP can directly perform operations upon the process state vector. Thus, although process state vectors are shared, they are only accessed by one processor at a time (either the IP to which the process is bound or the CP if the process is unbound).

The various operations on processes, *start*, *stop*, *kill*, *state*, and *wakeup*, are performed by *broadcasting* a message containing the process reference on the IPC bus. (A process reference is really a reference to the process state vector, except with a different type code. Type checking of invocations of primitive procedures prevents improper access to process state vectors.) If the target process is bound to an IP, that IP will accept the message (by matching the process reference) and perform the indicated operation. (If the IP is busy and unable to buffer the message, it will *refuse* the message, indicating to the requesting processor that it should resend the message at some later time.) Otherwise, no IP will accept the message, in which case it will be delivered to the CP, which will perform the specified operation on the process state vector. The CP may also modify its internal scheduling database; for example, if the process becomes runnable as the result of *start* or *wakeup*, then the process will be added to the CP's queue of runnable processes. During the transition between being unbound and being bound to an IP, the CP and the IP will *refuse* all messages directed at the process, causing the requests to be retransmitted until the transition completes and the new "owner" of the process begins accepting messages for the process. In this way, the bind and unbind transitions appear instantaneous to other processors, avoiding race conditions.

Unbinding a process is performed by the IP to which the process is bound. The unbinding is instigated either as the result of an operation performed by the IP (*block*, *wait*, *stop*, or *kill*), because the process terminated, or because the CP preempted the process (by sending a message to the IP). After the IP has updated the process state vector appropriately, it will notify the CP that it has unbound its process and is available to be bound to another process. If the process is runnable (it was preempted), then the CP will add it to its queue of runnable processes.

Each IP has a local memory which it uses in the interpretation of instructions. This local memory can also be used as a cache to reduce the rate of requests to the MM. For example, when a process is bound to an IP, the IP can read the process state from the process state vector into its local memory. In addition, it can read the top elements of the evaluation stack into its local memory, avoiding further access to the "real" stack object unless the stack changes greatly in size. It can also cache elements of immutable objects (particularly procedures). A possible organization for this cache is shown in Figure 9. The cache would be associatively searched on each instruction fetch. The local memory of an IP is intended merely as an optimization; its use must not affect the semantics of the machine language. Difficulties could arise in the case of shared, mutable objects, as side effects performed on a local copy of an object would not be visible to other IPs. Thus, we restrict the objects that may be wholly or partially copied into an IP's local memory to immutable objects (e.g., procedures), objects that are not shared (e.g., the evaluation stack), and objects whose sharing is specially-controlled (e.g., the process state vector).

When a process is unbound, the IP must write the process state back into the process state vector in the MM. It also must update the process stack in the MM by writing the "stack pointer" and all (changed) stack elements back into the corresponding objects in the MM. Encached contents of immutable objects can be retained, since all processes operate in the same address space.

When quiescence is established, the process state and the stack contents must be written into the MM, as described above. However, in addition, all IP caches must be cleared, so that the IP contains *no* references. During quiescence, the MM may reclaim some objects. Any references to a reclaimed object remaining in an IP cache would then be invalid. If the reclaimed object is later reused (a new object is created whose reference is identical to that of the reclaimed object), an IP could erroneously use old information in its cache to perform *fetch* operations on the new object. For this reason, all cache entries should be cleared at quiescence.

Figure 9. IP cache of immutable object elements.

object	element index	contents
vector#?1	0	bstring#n
vector#?1	1	bstring#m
vector#?2	0	vector#?3

5. The Implementation of Object References

The next three chapters are concerned with the implementation of storage objects. A storage object is one whose representation requires storage in addition to that provided by the object reference itself. In our simplified system, vectors are storage objects, whereas bstrings are not. For convenience, we will use the term object in these chapters to mean storage object.

We begin in this chapter by exploring the basic issues involved in implementing objects accessed via references. The next chapter discusses the implementation of automatic storage reclamation. Chapter 7 deals with storage allocation.

5.1 The Problem

When an operation such as vector *create* is called to create a new object, it allocates storage to hold the representation of the object, initializes the storage, and returns an object reference. This reference is a fixed length bit string that in some manner must identify the newly-created object so that when the reference is subsequently passed to other operations, it will be possible to locate the representation of the object. Determining the form of object references is a major design problem.

The problem is complicated by the use of a multi-level memory system. At any particular time, the object representation may exist in primary or secondary storage, or both. However, an object can be operated upon only while it resides in primary storage. When an operation is performed on an object, an object reference is passed to the operation. From that reference, the machine must be able to determine whether or not the object currently resides in primary storage.

If so, it must determine where the object resides in primary storage, so that it can perform the operation. Otherwise, it must determine where in secondary storage the object resides, so that it can copy the object into primary storage.

It is thus necessary to be able to map from object references to primary and secondary storage addresses. This problem is similar to those faced by conventional virtual memory systems. The major difference is that we are providing a single, unstructured address space consisting of a very large number of (mostly) small objects. Most conventional virtual memory systems provide relatively small, structured address spaces consisting of relatively large, fixed size pages. As we shall see, our implementation problems are much greater.

Because the primitive operations are so frequently performed, mapping from an object reference to its primary storage location must be fast. It is important that the mapping time be about the same as (and hopefully faster than) the actual primary storage access time. Other important considerations are the size of any data base needed to implement the mapping and the effects of the reference representation decision on paging performance and storage allocation.

5.2 Our Solution

After comparing various methods of implementing object references (reviewed in Section 5.5), based on our design goals and basic assumptions, we decided that object references should contain the physical address of the object representation in secondary storage. We assume that an object representation consists of a single, contiguous block of storage. If necessary, this storage can contain references to other objects, so no generality is lost. (This internal structuring would be transparent to the user.)

When a new object is created, space is allocated for the object in secondary storage. This secondary storage area is used to hold the object representation whenever the object is not resident in primary storage. The address of this storage is used as the data part of the object reference, which is returned as the result of the *create* operation.

An associative memory is used to map from the secondary storage address of an object in primary storage to its primary storage address. When an operation is performed on an object, the secondary storage address of the object is obtained from the reference and is looked up in the associative memory. If there is no matching associative memory entry, then the object is copied from secondary storage to primary storage and an entry is added to the associative memory. (The secondary storage address is already available, so no additional mapping is needed.) Otherwise, the primary storage address of the object is obtained from the associative memory entry.

The associative memory performs a function similar to a page map in a conventional virtual memory system. However, a conventional page map contains entries for all addressable pages, whether in primary or secondary storage. Our associative memory contains entries only for objects that are in primary storage. Nevertheless, because the average object size is likely to be quite small (perhaps only four words), the number of entries in the associative memory will be quite large. If each entry occupies two words, then the associative memory could be one-half as large as the primary storage itself.

Many conventional virtual memory systems provide address spaces that are either sufficiently small or sufficiently structured so that directly indexed tables can be used. For example, the Multics segmented virtual memory [5] is organized so that each addressable page can be identified by two small integers, a segment number and a page number within the segment. The page table entry for a page can be found by using the segment number as an index into the descriptor segment to obtain a page table, then using the page number as an index into the page table to obtain the page table entry.[1] This solution works because the number of segments used by any one process is relatively small, and the number of pages in each segment is also fairly small.

Our virtual address space, on the other hand, is very large and without internal structure. Thus, it would not be practical to use any form of directly-indexed page map. Instead, we must use a true associative memory. The practicality of our scheme depends upon the feasibility of building an associative memory that performs the desired mapping. We explore this issue in the next section.

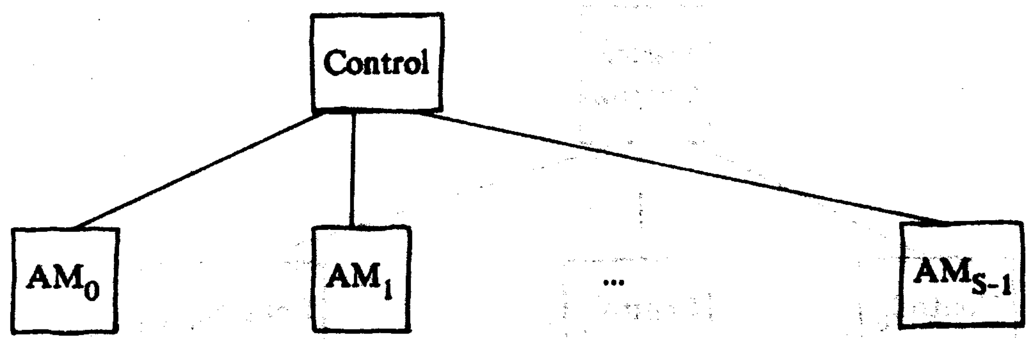
5.3 Associative Memory Design

Our associative memory contains one entry for each object currently residing in primary storage. Each entry contains the primary and secondary storage addresses of the object, plus a small amount of additional control information. Each entry thus occupies approximately two words. The size of the associative memory should approximate the average number of objects that can fit in primary storage. For a primary storage of 1 million words and an average object size of 4 words, the associative memory will contain 256K entries and occupy 512K words of additional storage.

Building a full associative memory of this size is impractical. Luckily, it is not necessary. The behavior of our associative memory can be closely approximated by a set associative memory [9], which is much easier to build.

[1]The actual mapping is more complicated, as the descriptor segment is itself paged.

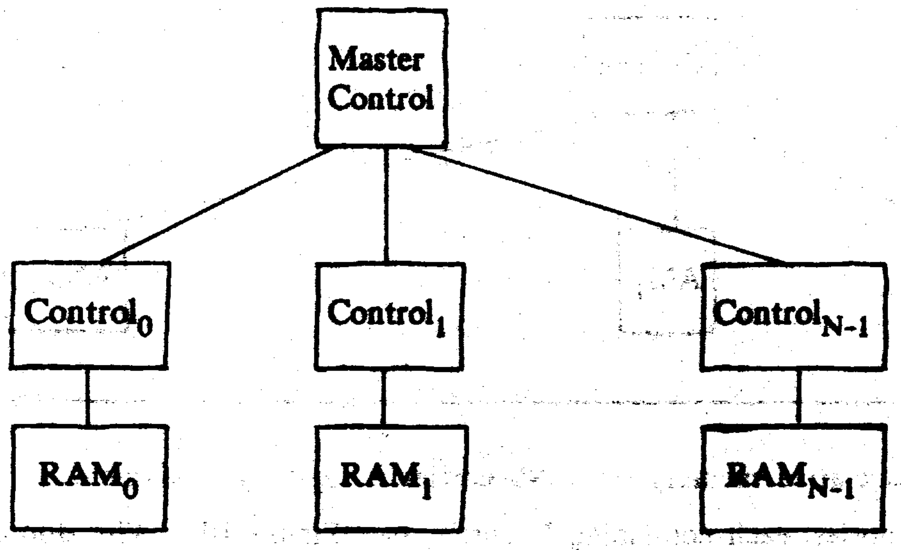
Figure 10. The logical structure of a set associative memory.



A set associative memory with NS entries is logically equivalent to S full associative memories, each containing N entries (see Figure 10). The domain of key values (in our case, secondary storage addresses) is partitioned into S *sets* by a hashing function h(k) that maps each key into a *set number* ranging from 0 to S-1. Each associative memory AM$_i$ holds the entries whose keys hash to *i*. To search for an entry given a key *k*, only AM$_{h(k)}$ need be searched. Searching a set associative memory in unit time thus requires only N comparisons, instead of NS for a full associative memory of the same size.

A set associative memory can be implemented using N ordinary random access memories (RAMs), each containing S entries, plus N associated controllers (see Figure 11). An entry with key *k* will be stored in a RAM at index h(*k*). To search for an entry with key *k*, each controller reads the entry at index h(*k*) from its associated RAM and compares its key with the given key. If the keys match, then the entry is returned to the master controller; the matching entry may be updated by subsequent operations. The advantage of the set associative

Figure 11. Set associative memory implementation.



organization is that a fast search can be performed, yet the amount of special purpose logic and the amount of parallel activity is only proportional to N, the set size.

The corresponding disadvantage of the set associative organization is that at most N keys from any particular set can be stored in the set associative memory at any one time. When a new entry is added to the set associative memory, if the corresponding AM_i is full, then one of the entries in AM_i must be removed to make room for the new entry. In our case, if more than N objects from one set are in active use, then a set associative map will produce more misses (page faults) than a fully associative map of the same size.

The expected miss rate is a function of the set size, N. A larger N will reduce the miss rate, but will also increase the cost of the associative memory. Smith [31] has shown that if the hashing function is sufficiently random,[2] then good (average) performance can be obtained with a small set size. In particular, Smith shows that for reasonably large associative memories using LRU replacement within each set, the set associative memory miss rate will be greater than the full associative memory miss rate by a factor of only N/(N-1). A set associative memory with a set size of 16 would therefore perform only about 7% worse than a full associative memory. We doubt that a set size of greater than 16 would ever be needed.

Assuming a set size of 16, a 256K entry set associative memory would be constructed out of 16 RAMs, each containing 16K entries of approximately 64 bits each. Using 16K x 4 bit memory chips, only 16 chips would be required for each RAM. Only 16 controllers are needed; these are sufficiently simple to be integrated on the RAM chips. Thus, the bulk of the set associative memory could be constructed using 256 identical chips.

Because of the large amount of memory required to implement the associative memory, we would expect it to be constructed using memory technology similar to that used for the primary storage. If so, then the lookup time will be approximately equal to the primary storage access time, resulting in a substantial overhead on each access to primary storage. However, this situation is really no different than in conventional virtual memory systems where page tables are stored in primary storage, and the solution is the same: use a fast translation lookaside buffer (TLB) [30]. A TLB is simply a small, fast associative memory used as a cache to speed up repeated accesses to recently used page map entries. It would be implemented using faster (i.e., more expensive) technology than the main associative memory, and could also be set associative. Of course, our TLB must be larger than a conventional TLB to obtain the same hit rate. Because we

[2]A random hashing function for secondary storage addresses can be obtained by exclusive-or'ing the low-order and high-order bits of the secondary storage address together. Because of the storage allocation method used (described in Chapter 7), just selecting either the low-order or high-order bits would probably not be sufficiently random.

are mapping smaller objects (compared to conventional pages), our working sets will contain more elements. How many more depends upon program behavior, but would be at most a factor of 100 (the ratio of the "page" sizes).

In summary, we conclude that an associative memory of sufficient size and speed can be built at an acceptable cost. The associative memory would be expensive, but only because it is large. Its cost could be one-half that of the primary storage, depending upon the average object size. However, to some extent, the large associative memory could "pay for itself," as we would expect the swapping of small objects to result in more effective use of primary storage, as only needed data will be swapped in, rather than whole pages.

5.4 Evaluation

The primary advantage of our method of implementing object references is that the mapping from an object reference to the current physical location of the object is fast. The mapping is performed by a single search of an associative memory. As described in the previous section, the associative memory can be implemented so that this search can be performed in a time no greater than the primary storage access time.

One important factor in the mapping speed is that the mapping data base is small enough to be stored entirely in fast memory. It can be stored in fast memory because it contains entries only for objects that currently reside in primary storage. If the map contained entries for every object in the system, then, because of the small expected average object size, the map would be comparable in size to the entire secondary storage. Such a large map, in addition to representing a large storage overhead, would also impose greater delays because of additional secondary storage accesses.

Another advantage is that object references can be compact. A reference size of 32 bits could provide 5 type code bits and 27 address bits, supporting 256M words of secondary storage (assuming object sizes that are multiples of 2 words). Because of the small expected average object size, the reference size will be close to the theoretical minimum.

Our method also has disadvantages. One disadvantage already mentioned is that its speed is obtained using a costly associative memory. Another disadvantage is that identifying objects by their secondary storage addresses makes it difficult to move objects in secondary storage.

The ability to relocate an object in secondary storage is useful in the implementation of secondary storage allocation. Whenever one attempts to allocate contiguous blocks of storage of different sizes from a single area, fragmentation can occur. Fragmentation occurs when a request for storage can not be satisfied, even though there is enough free storage available, because the free storage consists only of fragments that are smaller than the desired size. Fragmentation thus results in reduced utilization of the storage area. Fragmentation can be overcome if the allocated blocks can be moved, so that free blocks can be combined to form larger blocks.

Because we identify objects by their secondary storage addresses, moving an object in secondary storage requires either (1) that all references to the object be changed to reflect the new secondary storage address or (2) that information about the move be recorded so that subsequent accesses to that object will be redirected to the new secondary storage location.[3]

[3]One must also be sure that any new objects allocated in the old location will be distinguishable from the old object.

Finding all references to an object is difficult. In general, it requires a complete scan of the virtual memory (similar to that performed during garbage collection). Such a scan would be quite time-consuming. The redirection method is intended to be used as a temporary measure until all of the references can be converted, either by a garbage collection or by conversion upon use. (In this latter case, reference counts would be used to determine that all references had been converted.) However, the overhead of redirection, both in time and space, is likely to outweigh any benefit in increased secondary storage utilization obtained by moving objects.

Thus, our decision to identify objects by their secondary storage addresses will likely result in decreased secondary storage utilization, because of fragmentation. The issue of storage allocation and fragmentation is discussed further in Chapter 7.

5.5 Comparison with Other Methods

5.5.1 Capability Systems

A number of other mechanisms have been proposed or implemented that could be used to implement objects in our system. Most of these mechanisms were designed for systems using *capabilities* for controlling access to resources [20]. Many capability systems are similar to ours in that capabilities are used like references as a means of naming objects.[4] The primary functional difference is that most capability systems are designed to support the *explicit* deletion of objects, via explicit invocations of *delete* operations, as opposed to our automatic deletion of objects performed by the system after an object becomes inaccessible. Another difference is that most capability systems are designed under the assumption that objects are either large (e.g., segments) or special (e.g., I/O devices); thus, they can tolerate a larger overhead per object than we can.

One such system is the capability architecture proposed by Fabry [14]. In his system, each object is identified by a *unique identifier* (UID), which is a fixed length bit string guaranteed to be different than the UID of any previously created object. (UIDs could be obtained from a high resolution clock or an object creation counter that is never reset.)

Two mapping tables are used to implement this method. One table contains an entry for every existing object; the entry contains (at least) the current secondary storage address of the object. The other table is similar to our associative memory: it contains an entry for every object currently in primary storage; the entry contains (at least) the current primary storage address of the object. The primary storage map is consulted on each reference to an object. If no entry is found, the secondary storage map is consulted to locate the object in secondary storage.[5]

The primary motivation for identifying objects by UIDs is to support explicit object deletion. When an object is deleted, all entries for the object are removed from the maps. If the object is subsequently accessed (via a dangling reference), no entry will be found in either map (because UIDs are never reused). The reference is thus identified as dangling; an exception can be raised.

[4]This use of capabilities is called capability based addressing by Fabry [14]. Capabilities usually also contain *rights* that control specific kinds of access, e.g., read access and write access. Such rights could easily be added to our references.

[5]Fabry suggests that the primary storage map also contain secondary storage map entries for objects recently removed from primary storage. The idea is to reduce the number of accesses to the secondary storage map, which would be stored entirely in secondary storage.

Another advantage is that UIDs are distinct from secondary storage addresses, allowing complete freedom in moving objects in secondary storage to combat fragmentation. Other advantages of the indirection provided by the maps, e.g., implementing growing or shrinking objects, can be provided in our system using explicit indirection (one object containing a reference to another). In our system, however, indirection would be used only where needed.

The UID method shares our disadvantage of having a large primary storage map that must be consulted on every access to an object. In addition, however, the UID method requires a large secondary storage map containing an entry for every object in the system. If objects are as small and numerous as we predict, then this secondary storage map would occupy a significant fraction (e.g., 25%) of the secondary storage. More importantly, consulting the secondary storage map could easily double the number of secondary storage accesses.

Fabry discusses a number of other capability systems. Many of these do not support object references in their full generality. Others use two forms of capabilities, one a special form used only for capabilities in primary storage, with the goal of speeding up access or reducing the number of accesses to the primary storage map. Because our associative memory is adequately fast, we do not believe that the added overhead of converting between two forms of capabilities is justified.

5.5.2 Paged Systems

Like our system, most capability systems transfer individual objects between primary and secondary storage. The alternative, as exemplified by Bishop's ORSLA [7], is to transfer fixed-size pages. In ORSLA, an object reference contains the address of the representation of the object in a large, linear virtual address space. Bishop recommends a size of at least 2^{40} words for this virtual address space.

The virtual address space is implemented using a paged, multi-level memory, similar to conventional virtual memory systems. A primary storage map maps virtual page numbers into primary storage page numbers for each virtual page currently in primary storage. A secondary storage map maps virtual page numbers to secondary storage addresses.[6] The main difference between ORSLA's virtual memory and a conventional virtual memory is that ORSLA provides a single, very large virtual address space. Thus, the primary storage map is associative, rather than directly indexed. However, because of the large page size (e.g., 512 words), the primary and secondary storage maps represent a relatively small storage overhead.

As we explained in Chapter 1, a system that transfers fixed size pages between primary and secondary storage must somehow arrange things so that each page contains a reasonable amount of related information. Otherwise, primary storage utilization will be poor, extra secondary storage accesses will be needed, and secondary storage bandwidth will be wasted. ORSLA exploits spatial locality by allocating related objects in contiguous areas in the virtual address

[6]A secondary storage map is needed to allow the user to allocate virtual storage without allocating any corresponding physical storage. Bishop uses this feature to implement "unbounded" objects, e.g., stacks. A secondary storage map can also be used to turn a non-linear secondary storage address space (one with "holes") into a linear virtual address space.

space and by using a compacting single-area garbage collector. Note that even though ORSLA is paged, storage allocation is non-trivial, because small objects must be packed together on pages and large objects must be allocated in contiguous virtual pages. The garbage collector can move objects to combat fragmentation.

5.5.3 Very Small Pages?

We have chosen to investigate systems that do not need to exploit spatial locality. We are proposing a system that swaps individual objects between primary and secondary storage. However, given our assumptions about fast-access secondary storage devices, it is appropriate to consider the merits of a paged system with a very small page size that does not try to exploit spatial locality. Could such a system perform better than ours, or perhaps perform equally well but be simpler?

For the best comparison with our system, we will choose a page size approximately equal to the average object size, say 4 words. Thus, the primary storage map of this paged system will contain about the same number of entries as our associative memory, with entries only slightly smaller. We will assume that the secondary storage devices provide a natural linear, paged address space, so that no secondary storage map is needed.

One possibility is to implement each object as some integral number of pages, similar to a Multics segment. An object that fits in one page would be identified by the secondary storage address of that page. A larger object would be identified by the secondary storage address of a page table, which would contain the secondary storage addresses of the various pages comprising the object. The page table, if larger than a page, would itself be paged, etc. With such a small page size, the number of levels of page tables quickly becomes large for a moderate sized object, resulting in many extra secondary storage accesses.

The advantage of this scheme is that storage allocation is trivial. However, since the page size is approximately equal to the average object size, internal fragmentation due to rounding up each object to an integral number of pages is a major problem, as is the storage and time overhead of the page tables. For these reasons, we believe this scheme to be unworkable.

The alternative is to use a linear, paged virtual address space, allocating each object in a logically contiguous region of the virtual address space (not necessarily aligned on page boundaries). This system would be similar to ORSLA, except that (1) the page size would be much smaller and (2) no attempt would be made to group related objects together. An object would be identified by its virtual storage address, which would be equivalent to its secondary storage address.

This scheme reintroduces the storage allocation problem and fragmentation. In addition, it is likely to have poor paging performance. When an object smaller than a page is referenced and must be transferred to primary storage, the entire page must be transferred. However, the rest of the page is not likely to be useful to the program. Thus, for objects smaller than a page, this paged system will access secondary storage just as often as a system that transferred individual objects, but will make less effective use of primary storage. For objects occupying multiple pages, the paged system will most likely make many more secondary storage accesses than a system that transferred individual objects. Thus, using a paged, linear address space without utilizing spatial locality is not reasonable, even with a small page size.

6. The Implementation of Automatic Storage Reclamation

The secondary storage for an object is explicitly allocated by some primitive operation, such as the vector *create* operation. This storage can not be reclaimed as long as some process might perform an operation on the object. To maximize secondary storage utilization, the storage should be reclaimed as soon as possible after the object becomes inaccessible. It is up to the system to determine when an object becomes inaccessible and to reclaim the storage accordingly. How the system detects and reclaims inaccessible objects is the subject of this chapter.

6.1 Accessible Objects

Theoretically, an object can be reclaimed immediately after it is last used. In practice, we must define some notion of accessibility that can easily be implemented. The system must obey the property that any inaccessible object is guaranteed not to be used again, so that its storage can be reused to implement some other object.

All definitions of accessibility rely on the fact that an object reference is created only at the time the object is created; from then on, the reference can only be copied or destroyed. Thus, an object remains accessible only while there remain accessible references to the object. Usually there is at least one exception to this principle, namely, the *root* of the tree of objects. (The collection of objects actually forms a *graph*, or a tree with arbitrary sharing. We use the term *tree* for convenience.) The root object is always accessible, regardless of whether or not references to it exist. (We assume some primitive operation can construct a reference to the root whenever needed.)

Most definitions of accessibility define the set of accessible objects recursively as consisting of some set of immediately accessible objects, plus all objects that are components of accessible objects. In the case of a static tree of objects, there is a single immediately accessible object, namely, the root of the tree. However, in our system, which consists of a number of hardware modules storing and transmitting references, we must assume that any reference outside the actual stored tree of objects (e.g., in an IP) might be used. Thus, any object with a reference outside the stored tree of objects is considered immediately accessible. This definition places a heavy burden on the implementation. The implementation must keep track of every reference in every module and all references in transit between modules. If the system is in operation, then the implementation of storage reclamation must avoid any conflicts with normal operations that could result in references being overlooked.

The best solution to this problem is to design the system so that object accessibility is computed only when the system is in quiescence (see Section 4.3). In quiescence, no references exist except in the stored tree of objects in the MM. Under these conditions, race conditions with normal operations are impossible and the entire set of accessible objects can be computed looking only at objects in the MM.

6.2 Simple Garbage Collection

The simplest method for implementing automatic storage reclamation is the mark-sweep garbage collector [19]. A garbage collection is usually performed when an attempt to create an object fails because of insufficient free storage; however, garbage collection can also be invoked explicitly. In either case, normal system operation is suspended while the garbage collection is being performed. Thus, the garbage collector can compute the set of accessible objects by simply starting at the root and tracing the tree of accessible objects, marking each object seen. After this mark phase is completed, each accessible object has been marked. The garbage collector then examines each existing object: if the object is not marked, it is reclaimed. The set of existing objects is found by sweeping through the entire storage area.

A number of variations of the mark-sweep algorithm exist, differing primarily in their use of auxiliary storage. In addition, there are other garbage collection algorithms using more or fewer phases. However, all of these algorithms involve accessing at least every accessible object. In our system, accessing every accessible object would require a substantial number of secondary storage accesses. Although we are assuming a fast secondary storage, we are also assuming a large secondary storage. Thus, the time required to perform a garbage collection will be significant, making garbage collection unacceptable except at infrequent, scheduled intervals. (In Section 6.10 we suggest a garbage collector implementation that requires on the order of ten minutes to run. However, even this time is too long for an unscheduled interruption in service.)

The interval at which garbage collection must occur depends upon the rate of object creation and the desired secondary storage utilization. If the average secondary storage utilization is 80% (i.e., after garbage collection, 20% of secondary storage is unused), and the system creates objects at a rate of 5000 words/second (see Section 6.8), then a 100 million word system will run for only about an hour between garbage collections. Simple garbage collection of a large virtual memory is clearly inadequate.

6.3 Alternatives

There are a number of alternative methods of implementing automatic storage reclamation. One possibility is a parallel garbage collector, one that runs during normal system operation [32, 35, 13]. Because a parallel garbage collector does not require that the system be stopped, the garbage collection time and the garbage collection interval are less significant. In fact, a parallel garbage collector could be run continuously.

However, parallel garbage collectors have some disadvantages. The major disadvantage is that since the garbage collector runs in parallel with normal system operation, its operation is much more difficult to understand or prove correct [15]. Another disadvantage is increased overhead, because of memory contention and competing use of primary storage.

A related alternative is an incremental garbage collector, as proposed by Baker [2]. An incremental garbage collector distributes the garbage collection time by performing a small part of the garbage collection each time storage is allocated. Although the total garbage collection time is not decreased by this method, the disruption of normal activity is, since each interruption caused by performing more garbage collection is short. An incremental garbage collector avoids the complexity disadvantage of the parallel garbage collector, since normal activity is effectively stopped while the garbage collector runs. However, there are disadvantages. Performance degradation resulting from competing use of primary storage is still a problem. More importantly, the use of Baker's algorithm cuts secondary storage utilization by one-half. Such a performance penalty is reasonable only where uninterrupted service is essential. In addition, it is not clear how an incremental garbage collection algorithm can usefully be adapted to work in a multiprocessor system.

Another alternative is a single-area garbage collector, as in Bishop's ORSLA. Garbage collecting single areas is reasonable because, during any short interval, only small portions of the virtual memory will be modified. Only these areas are likely to contain much garbage. By focusing garbage collection on the active areas, the total long-term garbage collection time can be reduced. Furthermore, in ORSLA, only those processes using the garbage collected areas need be stopped during the garbage collection. Because garbage collecting a single area can be done in much less time than garbage collecting the entire virtual memory, stopping the processes will generally be acceptable.

To garbage collect a single area, the system must know which objects in the area are referred to from outside the area. (These are assumed to be the root objects of the tree of accessible objects in the area.) However, keeping track of inter-area references is complex. Also, as discussed in the Introduction, we prefer not to introduce areas into the visible machine architecture.

The fourth alternative, which we have chosen, is reference counts. We explore this alternative in the next section.

6.4 Reference Counts

The basic idea of reference counts is to associate a counter with each storage object to count the number of existing references to the object. When an object is created, a single reference to the object is created, and the reference count of the object is set to one. Whenever a reference to the object is copied, the reference count is incremented. Whenever a reference to the object is destroyed (e.g., by overwriting it with a new reference), the reference count is decremented. Whenever the reference count reaches zero, the object is inaccessible and can be reclaimed. (When an object is reclaimed, all contained references are effectively overwritten, causing the reference counts of the associated objects to be decremented. We assume that the system permanently contains a reference to the root object.)

One problem with the reference count scheme is that the reference counts of some inaccessible objects may not be zero, preventing those objects from being reclaimed. This problem results from cycles of references. If a group of objects contains a cycle of references, then each object in the cycle will have a reference count of at least one, although the entire group may be inaccessible. A similar problem arises if reference counts are limited in size. If a bounded reference count ever reaches its maximum value, it must remain at that value forever, lest the object be reclaimed prematurely.[1] For reasonable reference count sizes (e.g., 8 bits), this case will be relatively insignificant. We will use the term *cyclic garbage* to refer to all inaccessible objects not reclaimable by reference counts alone.

A system using reference counts thus requires garbage collection to reclaim cyclic garbage. However, because the rate of generation of cyclic garbage will in general be much less than the total rate of garbage generation, garbage collection in a system using reference counts can be much less frequent than in a system without reference counts.[2] We would anticipate garbage collection occurring at scheduled intervals on the order of once per day or once per week. With such intervals, stopping the system to perform garbage collection is probably

[1]These problems can be avoided in systems where all objects are immutable. Such systems can be designed so that cycles of references cannot be created. Furthermore, if the reference count of an immutable object becomes too high, one can always copy the object (producing a "new" reference) instead of copying the (old) reference, without any effect on the observable behavior of the system. A system with these characteristics has been designed by Weng [37].

acceptable, so that we can use a simple, mark-sweep algorithm. It would also probably be useful to combine the garbage collection with a salvaging operation that checks for errors in the file system.

6.5 Conventional Reference Count Implementation

Reference counts have not often been used to the extent we are proposing. The primary reason is that the conventional implementation of reference counts incurs a large overhead. Each time a reference to a storage object is copied or destroyed, a reference count must be updated. In a system like ours, these events occur at an enormous rate. Each assignment to a variable that denotes a storage object will cause a reference count operation. Every time a procedure is called, the reference counts of the argument objects must be incremented. When the procedure returns, the reference counts of the argument objects and the procedure's local objects must be decremented. It is easy to imagine a system spending 25-50% of its time updating reference counts.[3] Because garbage collection is needed anyway to reclaim cyclic garbage, it is difficult to justify the use of reference counts unless it *reduces* the overhead of performing automatic storage reclamation.

The other disadvantage of the conventional reference count implementation is complexity. In a system like ours, references are continuously being copied and destroyed in many system modules. Somehow, the net result of all this activity must be that every object has the correct reference count. A reference count too

[2]In a system using reference counts, a low rate of cyclic garbage generation is the mark of a well behaved program, just as a low rate of total garbage generation is the mark of a well behaved program in garbage collected systems. Mature programs will be tuned to minimize their rate of cyclic garbage generation.

[3]The Smalltalk-76 system, which implements reference counts in microcode on a minicomputer, is estimated to spend about 40% of its CPU time on reference count operations [18].

small could lead to the object being reclaimed prematurely, which could allow later errors to occur. A reference count too large will prevent the object from being reclaimed before the next garbage collection. In a system supporting concurrent operations, one must be sure there are no race conditions that could cause a reference count to transiently become zero, causing the object to be reclaimed prematurely. (An example of this problem is given in Section 6.7.)

These problems occur because the reference counts are counting every reference to an object anywhere in the system. The advantage of this approach is that the reference counts are continuously "valid": whenever a reference count becomes zero, the object is known to be inaccessible and can be reclaimed. The disadvantage, as we have seen, is that the cost and complexity of keeping reference counts continuously valid are high.

6.6 Queued Reference Counts

The alternative approach is not to count every reference in the system. Instead, our reference counts will count *only references stored as components of objects in the MM*. References outside the MM, or on their way in or out of the MM, will not be counted. (The root object will permanently have a non-zero reference count, regardless of the number of references to it stored in objects.)

One effect of this decision is to substantially reduce the number of events that cause reference count operations. *The only events that can cause reference count operations are those that change the contents of objects in the MM.* These events are simply the MM *store* request, which modifies the contents of an object, and object reclamation, which (in effect) destroys the contents of an object. (In our proposed architecture, the vector *create* operation creates an object containing *no* references to storage objects, so that object creation does not cause any reference count operations.) Note that the manipulations of references performed by processors now do not change reference counts. For example, operations performed on that part of the evaluation stack stored in an IP cause no reference count operations. Most accesses to procedure arguments and local variables will fall in this category.

Of course, if reference counts do not count all references to an object, we can not reclaim an object just because its reference count has become zero. After all, some references to the object may still exist in a processor and may later be used. To actually reclaim objects, we must force the system into quiescence. In a quiescent state, *all* references are stored as the contents of objects in the MM. Thus, during quiescence, our reference counts are valid and can be used to detect inaccessible objects.

One way to locate all objects with zero reference counts is to scan the entire memory looking at every object. However, a much better method is possible. The objects that we are looking for are objects whose reference counts have become zero since the last quiescent period. Therefore, whenever the reference count of an object x becomes zero (including when it is created[4]), we can add an entry **discard**(x) on a queue of suspected garbage GQ. At the end of the GQ cycle, when quiescence is next forced, every object with a zero reference count will have a **discard** entry on the GQ. The objects with zero reference counts can thus be identified by processing the GQ and checking the actual reference counts.

[4]When an object is created, a reference is created, but until (and unless) that reference is explicitly stored in the MM as a component of some object, the reference count of the newly created object is zero. It is necessary to add a **discard** entry to the GQ when an object is created so that the object can be located and reclaimed in the case where the reference is discarded without ever having been stored in the MM. This situation is likely to arise with transient objects whose references are stored only in the process stack cache in an IP and are discarded before the process is unbound.

While the above method is much faster than scanning the entire virtual memory, it requires the system to remain in quiescence until the set of reclaimable objects has been determined. Because a reference count can be zero at one time and then later be incremented (if the sole reference to the object is first outside the MM but later stored in the MM), some objects with entries on the GQ may not have a zero reference count. Determining the set of objects to be reclaimed thus requires examining the actual reference counts of the objects with entries on the GQ, which can be done only during quiescence.

The need to examine the actual reference counts of objects can be eliminated by adding an additional entry **resurrect**(x) to the GQ whenever the reference count of the object x goes from zero to a non-zero value. Together the **discard** and **resurrect** entries allow one to compute whether the reference count of an object was zero or nonzero *at the time quiescence was established*. If the last entry for an object in the GQ is a **discard** entry, then the reference count of that object must have been zero. If the last entry is a **resurrect** entry, the reference count was nonzero. If an object has a zero reference count during quiescence, then the object is truly inaccessible and can be reclaimed.

Using **resurrect** entries it is not necessary to keep the system in quiescence while the GQ is being processed. Instead, after quiescence is established, the old GQ is passed to a special GQ processor for processing. A new, empty GQ is created for future use, and normal system operation is resumed. Meanwhile, the GQ processor is processing the old GQ to determine which objects were inaccessible during the quiescent period. Any objects inaccessible then are inaccessible now and can be reclaimed. (If desired, the GQ processor can double check that the reference counts of these objects are indeed zero.) When an object is reclaimed, the reference counts of its component objects must be decremented. Any **discard** entries so produced must go on the *new* GQ, not the old one being processed.

Using this method, GQ processing is overlapped with normal system operation and the system is essentially uninterrupted. Putting the system in a quiescent state is similar to unbinding all running processes. In general, there will be some minimum rate of process switching needed anyway to maintain interactive response. As long as the GQ cycle time is longer than the process time quantum, there need be little performance degradation.

6.7 A Note on Ordering

In a conventional reference count implementation, it is essential that reference count operations be performed in the proper order. Reference count operations are partially ordered in that the increment operation caused by the creation of a new reference always occurs before the matching decrement operation caused by the destruction of that reference. If this ordering of operations is not preserved, a reference count could transiently become zero, which would cause the object to be discarded prematurely.

Consider the following example, and assume that our system uses a conventional reference count implementation. Suppose that the reference count of an object *v* is initially one. Then suppose that the following two operations are performed "concurrently" by two IPs:

```
vector$store (v1, i, v)
vector$store (v1, i, 0)
```

If the *store* operations are performed by the MM in the given order, then two reference count operations on *v* will be generated internally by the MM, one to increment v's reference count, and one to decrement it. If for some reason the second reference count operation is performed first, then the reference count of v will be zero for a short time. In a multiprocessor MM implementation such as the one described in Chapter 8, preserving the ordering of reference count operations requires additional synchronization.

Using the queued reference count scheme, it is not essential that reference count operations be performed in the correct order. All that matters is that the final reference count values at the end of each GQ cycle be correct. If the ordering of reference count operations is not preserved, however, then it is possible for a reference count to become transiently negative. The example above demonstrates this possibility, assuming that the reference count of v is originally zero (because the only references to v are in an IP).

The only change needed to allow the queued reference count mechanism to handle negative reference counts is to extend the increment and decrement operations to work for negative reference count values. Because negative reference counts are a transient condition, GQ entries need be generated only for transitions between zero and one. A minimum reference count value must be chosen. Decrement operations on a minimum valued reference count will be ignored; once a decrement operation is ignored, the object can be reclaimed only by the garbage collector. Because negative reference counts are not very likely (as the example above demonstrates, the situations that produce them involve race conditions in the user's programs), a minimum reference count value of -1 would probably be sufficient.

According to the description in the previous section, the order of GQ entries *is* important, as the decision on whether or not to reclaim an object was based on whether the *last* GQ entry for that object was a **discard** or a **resurrect** entry. There is, however, an alternative method of processing the GQ that does not depend on the ordering of entries.

The GQ entries for any particular object will consist of a sequence of alternating **discard** and **resurrect** entries, always beginning with a **discard** entry. If the object is accessible at the end of a GQ cycle, then there will be an even number of entries, ending with a **resurrect** entry. If the object is inaccessible, there will be an odd number of entries, ending with a **discard** entry. Thus, an alternative method for computing the set of inaccessible objects involves simply determining whether the number of GQ entries for each object is even or odd.

This computation can be performed making two sequential scans of the GQ and using two mark bits per object (see Figure 12). During the first scan, one mark bit is used to count (mod 2) the number of GQ entries for each object. At the same time, the other mark bit is used to detect multiple entries for an object; all but the first entry for each object is removed from the GQ (or overwritten). During the second scan, all objects whose first mark bit is on are reclaimed. Other objects have their mark bits reset. Removing duplicate entries is needed to avoid attempting to reclaim an object more than once.

The mark bits are stored in a header word associated with each vector. In addition to the mark bits, the header word will contain the reference count, size information, type information, etc. The mark bits are used only for GQ processing and garbage collection (described below); there is no interference with normal operation. If the mark bits were used during normal operation (e.g., to avoid placing duplicate entries on the GQ in the first place), then the system would have to remain in quiescence until the mark bits were reset to avoid

Figure 12. GQ Processing Algorithm.

```
process_gq = proc (gq: array[vector])
        % first scan
        for i: int in array[vector]$indexes (gq) do
                v: vector := gq[i]
                v.mark1 := ~v.mark1 % count mod 2
                if v.mark2 then % have seen before
                        gq[i] := undefined % remove entry from GQ
                  else
                        v.mark2 := true
                  end
                end
        % second scan
        for i: int in array[vector]$indexes (gq) do
                v: vector := gq[i]
                if v ~= undefined then % ignore overwritten entries
                        if v.mark1 then % odd number of entries
                                vector$reclaim (v)
                          else % reset bits for next time
                                v.mark1 := false
                                v.mark2 := false
                          end
                        end
                end
        end process_gq
```

interference. Note that using mark bits precludes concurrent processing of multiple GQs; a new GQ cycle cannot begin until the old GQ has been processed and the mark bits reset.

This method of GQ processing depends only upon the number of GQ entries and not on their order. It thus permits entries to be added to the GQ in any order. In addition, it eliminates the need to distinguish between **discard** and **resurrect** entries. This method might seem less robust than the originally proposed method, as the new method will make the wrong decision if any single GQ entry is lost, whereas the old method will make the wrong decision only if the last entry for an object is lost. However, we believe this difference to be insignificant, since most attempts to prematurely reclaim an object can be prevented by first checking that the actual stored reference count is zero.

We have shown that both reference count operations and additions to the GQ can be performed in any order. All that matters are the final reference count values and the total number of GQ entries for each object at the end of each GQ cycle. As we shall see in Chapter 8, this property allows significant freedom in the actual implementation. Removing ordering constraints reduces the need for internal synchronization in the MM and makes the system easier to understand. One less constraint simply means one less way to go wrong!

6.8 Queued Reference Count Performance

When the GQ processor reclaims an object, the reference counts of the component objects must be decremented. GQ entries produced by these reference count operations will be placed on the new GQ, which is not processed until the next GQ cycle. Thus, the reclamation of a tree of objects is performed breadth-first, one level per GQ cycle. A tree of depth N will require N GQ cycles to be completely reclaimed.

The maximum number of GQ entries that can be produced by the GQ processor while reclaiming a set of objects during one GQ cycle equals the number of references stored in those objects. (The worst case is where every object component is the sole existing reference to some storage object.) The need to use temporary storage can be avoided by storing these GQ entries in the reclaimed objects themselves. (If **discard** and **resurrect** entries are not distinguished, then a GQ entry is simply a reference.) The reclaimed objects can be chained together using their header words, which are separate from the actual object components.

It is important that the GQ cycle time be as short as possible without introducing excessive overhead from forcing the system into quiescence. A short GQ cycle time will minimize the sizes of the GQs, minimize the amount of primary storage wasted by inaccessible objects not yet reclaimed, and maximize the probability that the objects accessed by the GQ processor are still in primary storage.

Let us assume a GQ cycle time of 100 milliseconds, similar to current processor multiplexing intervals. How large will the GQs be? How much storage will be occupied by inaccessible objects?

We will assume a steady state where objects are created and destroyed at a rate of T objects per 100 milliseconds. We will also assume throughout this analysis that the probability of a reference count transiently becoming zero is negligible. A reference count can transiently become zero only if all references to the object are moved out of the MM, the MM copies of the references are overwritten, and a reference is later written back into the MM.

During each GQ cycle, T objects will be created by user processes and T objects will become inaccessible. Of the T objects created, some fraction $F(T)$ will become inaccessible during the same GQ cycle. For these objects there will be either 1 GQ entry (**discard**) or 3 GQ entries (**discard**, **resurrect**, and **discard**) generated, depending upon whether a reference to the object were ever stored in the MM. The remaining $T-F(T)$ new objects survive the cycle, producing 2 GQ entries (**discard** and **resurrect**).

The number of objects created before the current GQ cycle that become inaccessible during the cycle must also be $T-F(T)$ in the steady state. Each of these objects will generate a single discard entry. The remaining objects, created before the current GQ cycle and surviving that cycle, generate no GQ entries.

If we assume that objects created and discarded within the cycle generate 3 GQ entries (the worst case), then the number of GQ entries generated during each GQ cycle will be $3T$, independent of $F(T)$. The next question to ask is what the rate of object creation is likely to be. To answer this question, we measured the object creation rates of three CLU programs, running on a DEC PDP-10 (KA processor). (As in our proposed system, objects in this CLU implementation have header words, which are included in all measurements of object sizes.)

The first program measured was the CLU compiler, producing CLUMAC code (which consists of assembly language macro calls) for a fairly large source module. The compiler ran for 139 seconds of CPU time and created 26766 objects of total size 79777 words. The object creation rate was 192 objects/second (one object every 5.2 milliseconds) and 574 words/second. The average size of a new object was 3 words.

The second program measured was the CLUMAC assembler, assembling the CLUMAC output produced in the first step. The program ran for 165 seconds of CPU time and created 29637 objects of size 91637 words. The object creation rate was 180 objects/second (one object every 5.6 milliseconds) and 556 words/second. The average new object size was again 3 words.

To estimate the worst case, we wrote a program that did nothing but create objects, in this case PDP-10 words (which are two-word storage objects in this implementation). We ran this program for 24.1 CPU seconds, in which time it created 100059 objects, for a rate of 4150 objects/second (one object every 241 microseconds).

Thus, we find an object creation rate of approximately 200/sec for real programs and 4000/sec for a worst-case program. We suggest 1000/sec as a conservative estimate. For a more powerful system (faster and multiple processors), the estimates must be scaled up, for example to 2000/sec (real), 40000/sec (worst-case), and 10000/sec (conservative). We will base our analysis on this last figure, which corresponds to one object created and discarded every 100 microseconds.

For a GQ cycle time of 100 milliseconds, the GQ would contain 3 entries for each of 1K objects, a total of 3K entries occupying 3K words of storage. Two GQs (one being processed, the other being filled) would contain up to 6K entries, occupying 6K words. If the "average" depth of discarded structures is 5,[5] then there are effectively 5 generations of garbage structures in existence at any one time, for a total number of about 5K objects, occupying 15-30K words of storage. (All of this storage is virtual storage, but it is best that the objects remain in primary storage until they are reclaimed.)

We conclude, therefore, that a rate of 10 GQ cycles per second will provide adequate performance, in terms of the amount of storage needed to hold the GQs and the garbage objects. Of course, if the object creation rate is less than we have predicted, then the GQ cycle time can be increased accordingly. However, there is not much to be gained by increasing the GQ cycle time, since the

[5]It is difficult to estimate a number here short of constructing a CLU simulator that uses reference counts.

quiescent state is more or less equivalent to unbinding all processes, and processes are likely to be unbound at a rate no slower than once every 100 milliseconds for scheduling reasons.

The question of whether the GQ processor can reclaim garbage as fast as user programs generate garbage is discussed in Section 7.4. Should the GQ processor fall behind, and the GQs become "too large", the IPs would have to be stopped until the GQ processor caught up.

6.9 Garbage Collection

As described above, we have chosen to use the simple mark-sweep algorithm to perform infrequent, periodic garbage collection for the purpose of reclaiming cyclic garbage. This algorithm identifies all accessible objects by tracing the tree of accessible objects and marking each object seen. It then sweeps through the set of all objects, reclaiming all unmarked objects and resetting the mark bits. This algorithm assumes that all mark bits are initially cleared. However, if desired, a preliminary sweep can be made to verify that all mark bits are cleared before beginning the mark phase.

The primary function of the garbage collector is to reclaim cyclic garbage. In doing so, the garbage collector will be destroying references, some of which may point to accessible objects. The reference counts of those objects should be adjusted accordingly.

Using the normal reference count mechanism to adjust reference counts during garbage collection would probably not work, as both the garbage collector and the GQ processor could attempt to reclaim the same objects. Instead, the garbage collector should directly adjust the reference counts of *accessible* objects as references in garbage objects are destroyed, without generating any GQ entries.

However, a better solution is to have the garbage collector completely recompute all reference counts. In addition to reflecting changes resulting from cyclic garbage reclamation, this method will correct any erroneous reference counts resulting from hardware faults or system crashes, before additional damage can be caused.

A garbage collector that computes reference counts can be obtained by a simple extension to the standard algorithm: When an object is first marked, its reference count is set to one; when additional references to the object are seen during the mark phase, the reference count is incremented accordingly. Because the mark phase sees each accessible reference exactly once, this method will compute the correct reference counts. (The mark bit is not needed if a preliminary phase resets all reference counts to zero. Note that the standard mark-sweep algorithm can be viewed as a special case of this one, with the mark bit equivalent to a reference count whose maximum value is one.)

An implementation of the mark phase is shown in Figure 13. Here we assume only two types of objects, bstrings and vectors. The *vector_incr* operation manipulates the reference count and mark bit of a vector. The normal *size* and *fetch* operations can be used, as they do not cause reference count operations.

This mark phase algorithm requires temporary storage in the form of a stack containing the states of suspended procedure activations. The maximum size of this stack is proportional to the maximum depth of the tree of accessible

Figure 13. Recursive Mark Phase Algorithm.

```
recursive_mark_phase = proc (root: vector)
        vector_incr (root) % sets reference count of root to 1
        trace (root)
        end recursive_mark_phase

trace = proc (v: vector)
        % V is a vector whose reference count has just been set to 1.
        % Trace the references in V.
        n: int := vector$size (v)
        i: int := 0 % 'I' elements of V have been examined
        while i < n do % examine all elements of V
                e: any := vector$fetch (v, i)
                i := i + 1
                ve: vector := force[vector] (e) % test type of E
                   except when wrong_type: % if not a vector
                          continue % next iteration of loop
                          end
                if vector_incr (ve) % increment reference count
                   then trace (ve) % trace the component vector
                   end
                end
        end trace

vector_incr = proc (v: vector) returns (bool)
        % Increment the reference count of V.  If the new reference
        % count is 1, return true, indicating that the vector should
        % be traced.  Otherwise, return false.
        if ~v.mark then % if not marked
                v.mark := true % then mark it
                v.rc := 1 % set reference count
                return (true) % tracing is needed
                end
        v.rc := v.rc + 1 % increment reference count (unless at maximum value!)
        return (false) % tracing already started
        end vector_incr
```

---

objects, which is bounded only by the total number of accessible objects. Thus, in theory, this garbage collection algorithm could fail to operate because of insufficient temporary storage.

The need for temporary storage can be eliminated by storing the intermediate state of the computation in the tree of objects itself, a technique introduced by Schorr and Waite [29]. The state of each procedure activation consists of *v*, the object currently being traced, and *i*, the number of elements of *v* that have been traced.

The number of components *i* can be stored in *v* itself. As shall be explained in Chapter 8, during garbage collection the size field of *v* can be used to store the element counter *i*.

A reference to the vector *v* can be stored in the vector *v*[*i*] when tracing *v*[*i*], in whatever element of *v*[*i*] is currently being examined. The effect is to reverse the chain of references from the root to the node being traced. This reverse chain allows the procedure to find the proper node to return to when tracing of the current node is completed. An iterative version of this algorithm is presented in Figure 14.

In practice, it is probably best to combine these two methods, using a fixed-size stack and resorting to modifying the tree only when the stack becomes full. For most cases, a small stack (e.g. 1K elements) will suffice.

6.10 Garbage Collection Performance

The primary factor in the performance of the garbage collector is the delay caused by accessing secondary storage, as both phases of the garbage collector access most or all of the secondary storage. Compared to the secondary storage delays, computation time will be relatively insignificant.

In evaluating the mark phase, we will assume that the maximum depth of the tree is sufficiently small that the garbage collector stack will not overflow and all of the objects on the stack will fit in primary storage. By modifying the garbage collector code to occasionally touch all of the objects on the stack, we

Figure 14. Iterative Mark Phase Algorithm.

```
iterative_mark_phase = proc (root: vector)
        vector_incr (root) % sets reference count of root to 1
        v: vector := root % V is the vector currently being traced.
        vf: vector := root % VF is normally the father of V.
                % However, if VF=V, then V is the root.
                % This convention used to detect termination.
        while true do
                ve: vector := vector_current_element (v) % get vector element of V
                   except when none: % if all components of V have been traced
                        if vf=v then return end % finished tracing root
                        ve := v; v := vf % pop up one level
                        vf := vector_exch_ref (v, ve)
                          % store element back into V, obtain old VF from V
                        continue % next iteration of loop
                        end
                if vector_incr (ve) % increment reference count
                   then % we need to trace VE
                        vector_exch_ref (v, vf) % save father where son was
                        vf := v; v := ve % down one level in tree
                   end
                vector_skip_element (v) % done with that component
                end
        end iterative_mark_phase

vector_incr = proc (v: vector) returns (bool)
        % Increment the reference count of V. If the new reference count is 1,
        % set things up for tracing and return true, indicating that the vector
        % should be traced. Otherwise, return false.
        if ~v.mark then % if not marked
                v.mark := true % then mark it
                v.rc := 1 % set reference count
                save_size_info (v) % prepare to reuse size field
                v.size := 0 % initialize element counter
                return (true) % tracing is needed
                end
        v.rc := v.rc + 1 % increment reference count (unless at maximum value!)
        return (false) % tracing already started
        end vector_incr

vector_current_element = proc (v: vector) returns (vector) signals (none)
        % Return the first vector element of V not previously returned.
        % If none left, restore the state of V and signal none.
        % Causes no apparent side effects on V.
        size: int := compute_true_size (v) % compute size from redundant information
        while v.size < size do % look for first vector component
                e: any := v[v.size] % E is the component
                ve: vector := force[vector] (e) % test type of E
                   except when wrong_type: % if not a vector
                        v.size := v.size + 1 % proceed to next element
                        continue % next iteration of loop
                        end
                return (ve) % return vector component
                end
        % When loop terminates, v.size has been restored!
        signal none
        end vector_current_element

vector_exch_ref = proc (v: vector, e: vector) returns (vector)
        % Store E as the element of V just returned by vector_current_element.
        % Return the old value of that element.
        old: vector := v[v.size]
        v[v.size] := e % no reference count operations!
        return (old)
        end vector_exch_ref

vector_skip_element = proc (v: vector)
        % The element of V last returned by vector_current_element has been
        % processed. Update the state of V so that the next call to
        % vector_current_element will return the next vector element of V.
        v.size := v.size + 1
        end vector_skip_element
```

can ensure that all of the objects on the stack will remain in primary storage. In this way, we can guarantee that a secondary storage access can occur only going down the tree (away from the root!), and never when returning up the tree (towards the root).[6]

[6]It is not clear that this strategy is the best, but if not, it at least provides an upper bound on the number of secondary storage accesses. If our assumption about the sizes of the stack and the objects on it does not hold, then the upper bound will be less than a factor of two greater.

If all reference counts are one, then there is no sharing in the tree. Each reference examined will refer to an object not encountered previously. Because we can assume no relationship between the traversal order and the locations of the objects in secondary storage, each object examined will require an independent access to secondary storage. In this case, the number of secondary storage accesses will equal the number of objects. The number of objects (and thus the number of secondary storage accesses) can be predicted given the secondary storage size and the expected average object size.

If there is sharing in the tree, then a reference may refer to an object previously encountered. This object may or may not still be in primary storage when the subsequent access occurs. Because sharing requires cooperation, much sharing can be expected to occur within moderate sized subtrees, in which case the shared objects will remain in primary storage while the subtree is being traced. In the worst case, however, the number of secondary storage accesses will approximate the number of accessible references, which will be larger than the number of accessible objects.

To estimate the number of references in a system, we examined some CLU programs. We found the number of references to exceed the number of objects by about a factor of 2. We believe this number to be too high. Most of the sharing was in code (shared procedures) and related objects (linkage and debugging information). Data created by programs had lower ratios, generally less than 1.5.[7] In a large file system, we would expect data to predominate. For this reason, and because locality of reference *is* relevant here, we estimate that the number of secondary storage accesses will exceed the number of accessible objects by at most 25%.

[7]Measurements by Clark [8] on *data* in LISP programs showed almost no sharing of list cells.

In a system with 100 million words of secondary storage and an average object size of 4 words, there will be at most 25 million accessible objects and (we estimate) 30 million accessible references. If the average time to swap in an object from secondary storage is 100 microseconds, then a mark phase that performed 30 million secondary storage accesses in sequence would run for about one hour.

The performance of the mark phase can be greatly improved if multiple secondary storage requests can be processed concurrently. If multiple secondary storage requests are outstanding to a single secondary storage device, then the average access time can be reduced by processing requests in the proper order. The effective access time can also be reduced by using multiple secondary storage devices so that a number of transfers can proceed in parallel.

If enough concurrent requests can be generated, and there are enough secondary storage devices, the limit on the processing rate will not be the secondary storage access time, but will be the primary storage bandwidth and the overhead of initiating transfers. A factor of 10 improvement would involve transferring at most 40 words per 100 microseconds or 1 word every 2.5 microseconds, assuming an average object size of 4 words. This rate is similar to secondary storage transfer rates in current systems and should present no difficulty. The factor of 10 improvement could probably be obtained using 4 to 8 secondary storage devices, assuming approximately a factor of 2 improvement from reordering requests to each device. Careful consideration must be paid to designing the system to minimize the overhead of performing transfers. We refer the reader to a design by Ackerman for a high-throughput secondary storage module [1].

A factor of 10 improvement yields a mark phase time of only about 5 minutes. Obtaining this improvement depends upon generating large numbers of concurrent secondary storage requests. Fortunately, the amount of potential parallelism in the mark phase is practically unlimited. Tracing a vector involves tracing the components of the vector, all of which can be traced concurrently. The sequential mark phase algorithm can easily be modified to fork a new process to trace a particular vector whenever the total number of garbage collector processes is less than the desired number.

Such an algorithm is presented in Figure 15. This algorithm simulates multiple processes using an explicit state array. It uses a simple cyclic scheduling algorithm and explicit polling of pending secondary storage requests. In this way, we avoid many of the problems of implementing a general process mechanism, such as synchronization and interruptibility. This approach is possible because we believe that a single processor is sufficient to drive the secondary storage devices at the desired rate.

The only special operation used by the multiprocess mark phase algorithm is *vector_touch*. This operation will submit a secondary storage request for the vector, if needed. In any case, it returns immediately with an indication of whether or not the vector is currently in primary storage. *Vector_touch* is used to overlap computation with secondary storage accesses; when *vector_touch* indicates that a vector is not yet in primary storage, the mark phase will turn its attention to another process. *Vector_touch* is used only before accessing a new component of a vector being traced, as that is where secondary storage accesses are most likely. All other vector operations wait for any needed secondary storage accesses, as before.

Figure 15. Iterative Multiprocess Mark Phase Algorithm.

```
multiprocess_mark_phase = proc (root: vector)
        vector_incr (root) % sets reference count of root to 1
        maxn: int := 10 % maximum number of "processes"
        n: int := 1 % current number of "processes"
        i: int := 1 % the current "process"
        % The following two arrays store the process states.
        % Only the elements in the range 1..N are valid.
        v: array[vector] := array[vector]$fill (1, maxn, root)
                % V[i] is the vector currently being traced by process I
        vf: array[vector] := array[vector]$fill (1, maxn, root)
                % VF[i] is normally the father of V[i]. However, if VF[i]=V[i],
                % then V[i] is the root of the subtree being traced by process I.
                % This convention used to detect termination.
        while true do
                ve: vector := vector_current_element (v[i]) % get vector element of V[i]
                  except when none: % if all components of V[i] have been traced
                        if vf[i]=v[i] then % finished tracing root of subtree
                                if n=1 then return end % all processes done
                                v[i] := v[n]; vf[i] := vf[n]; n := n-1 % delete process
                                i := i+1; if i>n then i := 1 end % select new process
                                continue % next iteration of loop
                                end
                        ve := v[i]; v[i] := vf[i] % pop up one level
                        vf[i] := vector_exch_ref (v[i], ve)
                          % store element back into V[i], obtain old VF[i] from V[i]
                        continue % next iteration of loop
                        end
                if ~vector_touch (ve) then % VE not in primary storage
                        i := i+1; if i>n then i := 1 end % select new process
                        continue % next iteration of loop
                        end
                if vector_incr (ve) % increment reference count
                  then % we need to trace VE
                        if n < maxn then % fork a new process
                             n := n+1; v[n] := ve; vf[n] := ve
                          else
                             vector_exch_ref (v[i], vf[i]) % save father where son was
                             vf[i] := v[i]; v[i] := ve % down one level in tree
                          end
                  end
                vector_skip_element (v[i]) % done with that component
                end
        end multiprocess_mark_phase
```

The sweep phase will always perform at least as well as the mark phase. The sweep phase examines every object in the virtual memory. However, it can examine the objects in any order, in particular, in whatever order will minimize the access time. If there are multiple secondary storage devices, it can sweep each device concurrently. If we assume that the sweep phase takes the same time as the mark phase, then the total time of the garbage collection will be about ten minutes. Clearly any time of this magnitude is acceptable for infrequent scheduled garbage collections.

6.11 Evaluation

Our proposed mechanism using queued reference counts and occasional garbage collection has a number of advantages. Because reference counts will detect most garbage, garbage collection need occur only infrequently at scheduled intervals. At such intervals, stopping the system to perform garbage collection is often acceptable. We can thus use a relatively simple garbage collection algorithm, which can easily be modified to perform a salvaging function. In addition, because we can devote the entire resources of the system to performing the garbage collection, the garbage collection time can be quite short.

The queued reference count scheme has both simplicity and performance advantages. The set of counted references is well defined. The set of events that can change the reference counts is easily derived. Problems of race conditions and synchronization to preserve ordering are avoided. The rate of reference count modification is substantially reduced, and most reclamation activity is performed concurrently with normal processing. The overhead of using reference counts is thus minimized. The one functional disadvantage of queued reference counts is that the reference counts cannot easily be used by primitive operations or user programs, for example, to allow the representation of unshared immutable objects to be modified (without first being copied). This disadvantage would be serious in a system that supported only immutable objects, e.g. a data-flow architecture [11].

The queued reference count scheme is similar in strategy to a mechanism proposed by Deutsch and Bobrow [12]. That mechanism was designed for a standard single-process LISP system. References in the evaluation stack are not counted, thus reducing the number of reference count operations. Unlike our mechanism, the reference counts are not stored in the objects, but in separate tables. These tables are arranged so that no storage is required for reference counts whose value is one, the most common case. The user process does not directly access reference counts. Instead, *all* reference count operations are queued in a transaction file (TF) and processed in batches. When the TF is processed, a copy of the current evaluation stack is given to the transaction file processor (TFP) so that references in the evaluation stack will be considered. This action of passing the current evaluation stack and TF and creating an empty TF for future use is equivalent to the quiescent state in our mechanism. Once the TFP has begun processing the TF, the user process can be resumed. After the TFP has finished processing the TF, the reference count tables will contain the true reference counts at the time the TFP was started. The TFP can reclaim any object that has a zero reference count and is not referred to by the evaluation stack. Because the TFP has sole access to the reference count tables, it can directly perform any reference count operations resulting from object reclamation. It can thus discard entire structures at once (if desired), rather than one level per cycle. The disadvantage of this mechanism is that TF entries are generated for every reference count transaction, not just those between zero and one. The number of additional TF entries will depend upon the amount and activity of shared objects.

7. The Implementation of Storage Allocation

There are actually two storage allocation problems, secondary storage allocation and primary storage allocation. Secondary storage is allocated as part of the vector *create* operation; secondary storage is deallocated when a vector is reclaimed (by the GQ processor or by the garbage collector). Primary storage is allocated when a vector is created or swapped in; primary storage is deallocated when a vector is removed from primary storage (as part of reclamation or to make room for other vectors). We will begin by considering secondary storage allocation, which is a more serious problem.

A note of terminology: Because the units of storage allocated and deallocated correspond to the units of information transferred between primary and secondary storage, we will call these storage units (primary storage or secondary storage) *pages*. Unlike most conventional paged systems, these pages come in many different sizes.

7.1 Secondary Storage Allocation

The secondary storage allocator must satisfy a number of constraints. First, it must be prepared to satisfy arbitrary requests for storage ranging from one word to some maximum value chosen by the system designer. In response to a request, it must allocate (at least) the desired amount of contiguous secondary storage.

Second, most allocated pages cannot be relocated in secondary storage without stopping the system. The secondary storage allocator thus cannot depend upon being able to perform compaction, except as part of periodic system maintenance. One should expect that free storage will be scattered throughout secondary storage. It is important that storage underutilization caused by fragmentation be controlled. This problem is especially serious because compaction can not be used.

Note that with respect to a single system, storage underutilization is meaningful only when a request for storage fails. Storage is underutilized if free storage exists but cannot be used to satisfy the failing request. Alternatively, the degree of storage underutilization will determine how much secondary storage is needed to allow a given computation to be performed. In this case we are in effect comparing multiple systems with different storage sizes to find the smallest system that can perform the computation.

Finally, because secondary storage allocation is performed frequently, it is important that allocation and deallocation be fast. Here we are concerned primarily with minimizing the number of secondary storage accesses, although processor time is also important. The free secondary storage pages will likely be identified by chaining them together on free lists. We must reject any method that involves searching through free storage lists to perform allocation or deallocation, because such searching would likely incur multiple secondary storage accesses. Many standard storage allocation algorithms involve searching a free list to find a free page large enough to satisfy a request or to find adjacent free pages to merge with a page being deallocated. Such algorithms are not acceptable.

7.2 Zoned Allocation

Given the above constraints, we conclude that the proper strategy is to divide secondary storage into a number of *zones*, each of which provides pages of a single size.[1] (For the time being, assume that the number of zones (page sizes) equals the number of possible vector sizes and that the maximum vector size is sufficiently large that the overhead of using multilevel structures is insignificant.) Within each zone, storage allocation can be handled much as in a conventional paged system. The available secondary storage address space can be split up into pages, each of which is either allocated or free. The free pages can be chained together on a free list. Allocation and deallocation require at most one secondary storage access to read or write the free list pointer in the page being allocated or deallocated. Within each zone, there is no fragmentation, as all pages are the same size.

While there is no fragmentation from the point of view of each zone, there can still be storage underutilization from the point of view of the system as a whole. If a particular zone becomes full, it is possible that a request for storage will fail even though free storage exists in other zones. If free storage exists in a zone that provides larger pages, then a larger page can be used (at a cost of introducing internal fragmentation). Ultimately, however, a request will arrive when free storage exists only in zones providing smaller pages. Unless the relative zone sizes can be adjusted (unlikely if allocated pages cannot be moved) or there exist contiguous free pages that can be combined to form a page of sufficient size (ultimately unlikely, and probably an undesirable solution in any case, as it adds complexity and defeats our method of determining the size of a page from its reference), the request will fail. Any free storage in other zones will be useless.

The amount of storage underutilization resulting from partitioning secondary storage into zones depends on the variation over time of the distribution of the sizes of allocated pages. Each zone must be large enough to handle the peaks in the number of allocated pages of the corresponding size. If

[1]The term *zone* is borrowed from the Smalltalk-76 implementation, which uses a similar secondary storage allocation method.

the size distribution remains relatively constant, then each zone will have similar utilizations. Should a zone become full, the other zones will be nearly full, so that the amount of wasted storage is small. On the other hand, if the size distribution varies wildly over time, then the utilization of the most utilized zone at any one time will likely be much greater than the utilization of the other zones at that time, resulting in significant storage underutilization.

The amount of storage underutilization will in general increase as a function of the number of zones. There are two reasons for this relationship. One reason is that the relative effect of the one full zone (the one rejecting a request for storage) is greater for smaller numbers of zones. If there are only two equally sized zones, then the storage underutilization can not be more than 50%, regardless of the variations in the utilizations of the two zones. The worst case storage underutilization for ten equally sized zones is 90%.

The other reason is that reducing the number of zones will tend to smooth out the time variations in the distributions of allocated page sizes. If the number of zones is reduced, some requests that would have gone to different zones will now go to the same zone. Variations in the numbers of pages previously allocated from the different zones will tend to cancel out when the zones are merged, resulting in less variation in the overall distribution of page sizes.

There are two ways to reduce the number of zones (the number of page sizes), each of which has an associated cost. One way is to reduce the maximum page size. The cost of this reduction is an increase in the number of objects that must be represented by multi-level structures, which will increase the amount of storage occupied by "page tables". For four CLU programs measured, we found the percentage of objects larger than 128 words was less than 1% for each program. Thus, for maximum page sizes of at least 128 words, the overhead of page tables will be insignificant.

The other way to reduce the number of zones is to provide only some of the possible page sizes in the range from two words (one header word plus one data word) to the maximum page size. Requests for other page sizes in this range would be satisfied by allocating pages of the next larger supported size. The cost of this method is that it introduces internal fragmentation, storage that is wasted because it is allocated as part of an object but never used. The amount of internal fragmentation can be controlled by proper selection of the supported page sizes. For example, one could provide page sizes that are powers of two. For the same four CLU programs, we found that providing the powers of two from 2 to 128 words would result in 17-19% internal fragmentation. Better results can be obtained by matching the page sizes to the expected size distributions. For these programs, adding page sizes of 3, 96, 5, and 48 words would reduce the internal fragmentation to about 7%.

In a real system, storage is used for a number of purposes. Most storage will probably be used for "file" storage that changes at a relatively slow rate. Other storage is created and discarded at a relatively rapid rate by processes. The total storage usage is the sum of all activities. In a multiprocess, multiuser system, we would expect a relatively stable, slowly changing distribution of allocated page sizes. A slowly changing distribution can be handled by adjusting the relative zone sizes as part of periodic maintenance. What can't be handled in this manner are the dynamic ups and downs caused by the activities of processes. For example, a particular program may create a large number of objects of a particular size, which are all discarded when the program terminates. These variations can be handled only by providing extra storage in each zone. We would expect the variations caused by running processes to be a small percentage of the total storage usage, so that the amount of storage underutilization will be tolerable.

7.3 Block Allocation

To implement a number of storage zones, we will divide secondary storage into fixed size *blocks*. Each block will be assigned to a particular zone and will therefore provide a single page size. The block size will probably be a multiple of the maximum page size, chosen to minimize the amount of storage wasted when a block is carved up into pages of any of the supported page sizes. The block size may also be affected by the addressing characteristics of the secondary storage devices, as we require contiguous secondary storage addresses within each block. Blocks will be identified by block numbers, whose choice will again be related to the addressing characteristics of the secondary storage devices. (However, for each secondary storage device, the block numbers should be reasonably compact, to allow the use of device tables indexed by block number.) A secondary storage page will be identified by a device number, a block number on that device, and an offset within the block.[2] (This secondary storage "address" will be used as the data part of a vector reference.)

Each secondary storage device will have an associated table in fast storage mapping block numbers to zone numbers. (A directly indexed table requires only 4 bits per secondary storage page, assuming 16 or fewer zones.) The tables can

[2]Alternatively, the page number within the block can be used instead of the offset. This method saves one bit of address length where odd page sizes are used, assuming a minimum page size of two words.

be used to determine the size of a secondary storage page given its address. This ability is needed so that the proper amount of primary storage can be allocated before the contents of a page are transferred from secondary to primary storage.

Two methods can be used to assign blocks to the various zones. Using *static* assignment, the expected distribution of page sizes is determined in advance (e.g., by measuring existing programs), and blocks are assigned to zones accordingly during system initialization. Using *dynamic* assignment, all blocks start out *empty* and not assigned to any zone. When a page of a given size must be allocated and there are no free pages of that size, an empty block is assigned to the corresponding zone and is split up to form pages of the desired size. The ultimate zone sizes thus will reflect the actual page size distribution, rather than a predicted page size distribution.

Additional flexibility can be gained using dynamic assignment if blocks are removed from zones when the blocks become empty. (This ability requires the free lists to be organized so that each block in effect has its own free list, to allow the free list to quickly be adjusted when a block is removed from a zone.) Dynamic unassignment permits some adjustments in zone sizes in response to changing size distributions. The effectiveness of dynamic unassignment depends upon how many blocks become empty when the number of allocated pages in a zone decreases. The probability that a given block will become empty decreases rapidly as the number of pages per block increases. Dynamic unassignment is thus most helpful when storage usage shifts from large pages to small pages and least helpful when storage usage shifts from small pages to large pages. For dynamic unassignment to be useful, the block size should be minimized.

Dynamic unassignment does not solve the problem of storage underutilization, but it does offer some help in recovering from a storage unbalance. Nevertheless, the best strategy is to make sure there is enough free storage in each zone to handle anticipated usage. An obvious source of difficulty here is the possibility that a program will maliciously or erroneously allocate a large number of objects of a particular size. Here dynamic assignment and unassignment perform worse than static assignment. Static assignment will stop the program when the particular zone becomes full. If the objects can be reclaimed, then the system can resume normal activity without much disruption. Dynamic assignment and unassignment will allow the program to run longer, possibly reducing the free storage levels in all zones to near zero. Furthermore, even if the program is stopped and the objects reclaimed, there is no guarantee that the blocks that were dynamically assigned to the zone can be emptied and so be made available again to the other zones. It is likely that these blocks will also contain a small number of objects created concurrently by other processes. These objects may prevent the storage unbalance from being corrected.

The solution to this problem and others like it must be a resource allocation mechanism that limits the ability of individual users to obtain more than an appropriate share of the system's resources. We will examine this issue in Chapter 9.

7.4 Evaluation

In this section we evaluate the block allocation method with respect to the two criteria of storage underutilization and allocation/deallocation speed.

There are two sources of storage underutilization. One is internal fragmentation, resulting from rounding up request sizes to the next supported page size. Our evidence indicates that a proper choice of about 11 page sizes matched to the expected object size distribution can limit storage underutilization caused by internal fragmentation to under 10% of the total secondary storage.

The other source of storage underutilization is external fragmentation resulting from partitioning secondary storage into zones. The magnitude of this underutilization depends upon the amount of short-term variation in the distribution of object sizes. We have no basis for making specific estimates, but we believe that in most cases external fragmentation will be a small percentage of the total secondary storage size.

It should be noted that conventional file systems also have fragmentation problems, particularly internal fragmentation resulting from allocating files as integral numbers of fairly large pages or blocks. The large page size, plus the overhead of directory entries, makes large numbers of small files prohibitively expensive. Where small objects are desired, our system will utilize secondary storage more efficiently than conventional systems.

With respect to allocation and deallocation speed, the block allocation method requires at most one secondary storage access per allocation and deallocation. Is this good enough? We have conservatively estimated an object creation/reclamation rate of 10000 objects per second. If a secondary storage access requires an average of 100 microseconds, then performing 20000 secondary storage accesses in sequence would require two seconds! At this rate, every second of user computation would cause two seconds of secondary storage access delay, limiting the system to at best 50% of intended capacity.

There are a number of reasons why things are not really this bad. For one thing, many secondary storage requests can be performed concurrently. Object reclamation is performed concurrently with normal system operation. The GQ processor can easily be designed to submit multiple deallocation requests in parallel. Storage allocation is performed by multiple processes, so there will likely be concurrent allocation requests. By having separate free lists for each secondary storage device and by reordering multiple requests to each device (see Section 6.10), we can process requests in parallel much faster than in sequence.

Another factor is that the secondary storage access performed during allocation serves only to obtain the new free list pointer from the allocated page. The allocation operation can thus "return" the address of the newly allocated page immediately, without waiting for the secondary storage request to complete (as long as the secondary storage request is guaranteed to be performed before any other operation on that secondary storage page). Thus, the requesting process will not be delayed unless it submits another allocation request to the same device before the secondary storage operation completes. In this manner, much of the secondary storage access delay can be overlapped with user computation.

Finally, because much of the allocation and deallocation activity results from the continual creation and reclamation of transient objects, the number of secondary storage requests can be reduced substantially by maintaining the "top" portion of the free list in fast storage. When the rates of object allocation and deallocation are in (short-term) balance, most secondary storage page allocation and deallocation requests will be handled without any secondary storage accesses.

7.5 Primary Storage Allocation

Compared to secondary storage allocation, primary storage allocation is trivial. When allocating primary storage, we do not have to be concerned with minimizing the number of accesses to storage. Furthermore, because objects are not fixed for all time in specific primary storage locations, fragmentation is much less of a problem. Any of the traditional storage allocation algorithms can be used. To keep things simple, we suggest using the same zoned allocation scheme for primary storage. Because primary storage usage reflects short-term program behavior, the zone sizes will occasionally need adjusting. Adjusting primary storage zone sizes is easy: one can swap objects out to secondary storage to free up blocks for reassignment. At worst, one could simply swap out all objects and start from scratch.

8. System Structure: the Memory Module

The purpose of this chapter is to present an actual design for a memory module based on the ideas presented in the previous three chapters. The design will consist of a collection of hardware modules and interconnections. The functions of the modules will be described, as will the various kinds of messages transmitted between modules. Particular attention will be paid to questions of synchronization and flow control.

8.1 Vectors and Pages

The previous chapter introduced the notions of primary and secondary storage pages. At this point, it is convenient to introduce a third kind of page, called a *virtual page*. All pages are structured, mutable "objects" containing fixed numbers of elements. Unlike true objects, pages are explicitly deallocated, via *dealloc* operations. Thus, in some sense it is improper to call a page an object; however, for convenience we will continue to do so. In addition, as described in the previous chapter, pages come in only a few different sizes.

A secondary storage page resides in secondary storage and is identified by its secondary storage address. A primary storage page resides in primary storage and is identified by its primary storage address. A virtual page resides in virtual storage, that is, it normally resides on secondary storage but will be copied into primary storage as necessary to support fast access. Virtual pages are the basic "objects" provided by a virtual (multilevel) memory.

Virtual pages are implemented using secondary storage pages and primary storage pages. Each virtual page is represented by a secondary storage page, which provides the "long-term" storage for the contents of the virtual page. A virtual page reference will be equivalent to the corresponding secondary storage page reference, that is, it will contain a secondary storage address. When a virtual page is being used, its contents will be temporarily stored (cached) in a primary storage page. The correspondence between virtual pages and primary storage pages is maintained by a page map.

Vectors will be implemented using virtual pages, henceforth simply called pages. Each vector will be represented by a single[1] page. Element 0 of the page will be the vector header word, which is a bstring value composed of a number of fields, described in Figure 16. The remaining elements of the page will store the elements of the vector. A vector reference will in effect contain the reference of the page that represents the vector. In particular, the vector reference will contain the secondary storage address of the page. The type code will indicate that the reference is a vector reference. (No *page* type code will be

---

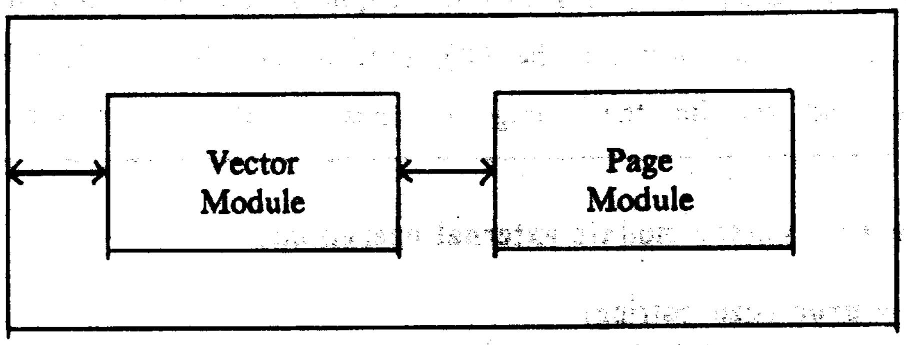
Figure 16. Fields of a vector header word.

type code	Identifies the page as being allocated and as representing a vector.
ref count	Counts the number of references to the vector in the graph of objects in the MM.
mark bit 1	Used during GQ processing to count (mod 2) the GQ entries for this vector. Used during the garbage collection mark phase to mark accessible vectors.
mark bit 2	Used during GQ processing to identify duplicate GQ entries for this vector. Used during the garbage collection mark phase to indicate that the vector size is one less than the page size (the page is "full").
size	Normally, the number of elements in the vector. During the garbage collection mark phase, the number of vector elements that have been examined.

stored in a vector reference, as the representation type is implied by the abstract type.) Only the vector implementation will be allowed to convert between a vector reference and the corresponding page reference.

In section 6.9, it was mentioned that the vector size field in the header word can be used as an element counter by the garbage collector. This "trick" depends upon the ability to determine the size of a secondary storage page (and therefore a virtual page) given its secondary storage address (see Section 7.3). If we know that the page representing a vector is full, meaning that the vector size is equal to the page size less one (the header word), then the vector size can be computed directly from the page size. Otherwise, the vector size can be stored in the last word of the page while the vector is being traced. All we need to do is be able to tell whether the page is full or not. We can use the second mark bit in the vector header word for this purpose.

Figure 17. Memory module structure.



¹If the memory module supported large or dynamic objects, then such an object could be represented by a top-level page containing references to component pages. Using pages (rather than vectors) for the lower-level components is possible because the entire structure is viewed as a single object outside the MM; it will be discarded as a unit. Thus, separate reference counts for the internal components are not needed.

The distinction between vectors and pages allows a similar division in the design of the MM. The MM will be split into two modules, a *page module*, which implements pages, and a *vector module*, which implements vectors in terms of pages. This structure is shown in Figure 17. The page module performs storage allocation, reference mapping, synchronization, and implements the multi-level memory. The vector module implements automatic storage reclamation. This separation of function helps to improve the organization of the design, making it simpler and more understandable.

8.2 Vector Module Specification

The explicit function of the vector module is to implement the primitive vector operations described in Chapter 3. For convenience, these operations are listed again in Figure 18.

In addition, the vector module has one implicit function: it must automatically reclaim the storage of objects that have become inaccessible. As described in Chapter 6, this function requires that the vector module update reference counts, construct the GQ, process the GQ to identify inaccessible objects, and reclaim the storage of those objects. The vector module is

Figure 18. Vector module external operations.

```
create = proc (size: bstring)
                returns (vector)
                signals (negative_size, size_too_large, no_storage)
equal = proc (v1, v2: vector) returns (bstring)
size = proc (v: vector) returns (bstring)
fetch = proc (v: vector, index: bstring) returns (any) signals (bounds)
store = proc (v: vector, index: bstring, element: any) signals (bounds)
```

responsible for initiating the quiescent state and coordinating the establishment of quiescence with the processing module. The vector module is also responsible for performing garbage collection, at the request of the control processor.

In performing storage reclamation, a number of "internal" operations are performed upon vectors by the vector module itself. These operations were described in Chapter 6 and are listed in Figure 19. (For convenience, we assume only one mark bit, rather than two.) For the remainder of this chapter, we will not distinguish between external and internal operations.

The specification of the vector operations requires that the operations be atomic, which means that any sequence of vector operations performed concurrently must produce a result equivalent to performing the operations in some order. Actually performing only one operation at a time would guarantee atomicity, but would be unnecessarily inefficient. Some operations will take a relatively long time to perform, because substantial processing is being done (e.g.,

---

Figure 19. Vector module internal operations.

```
incr_rc = proc (v: vector) returns (bstring)
decr_rc = proc (v: vector) returns (bstring)
reclaim = proc (v: vector)
mark = proc (v: vector)
unmark = proc (v: vector)
marked = proc (v: vector) returns (bstring)
touch = proc (v: vector) returns (bstring)
incr = proc (v: vector) returns (bstring) % used by trace phase
current_element = proc (v: vector) returns (vector) signals (none)
exch_ref = proc (v: vector, e: vector) returns (vector)
skip_element = proc (v: vector)
```

create), or because data must be transferred between primary and secondary storage. We would like other operations to proceed in parallel with these long operations.

Allowing concurrent operations requires us to analyze the effects of concurrent operations to see where explicit synchronization is needed. Here, we are primarily concerned with the effect of two operations being performed on the same vector. (Other conflicts, such as concurrently creating two vectors, involve simultaneous access to internal data bases; these synchronization problems are discussed in the description of the page module.) We can immediately eliminate all of the garbage collector operations, on the assumption that we are using the pseudo-parallel implementation (see Figure 15), which performs all vector operations in sequence. The results of an analysis of the remaining operations is presented in chart form in Figure 20. This chart includes one entry for each combination of operations. A letter indicates that there can be no conflict between the two operations; a number indicates a possible conflict.

As the figure shows, there are only five kinds of potential conflicts. These potential conflicts can be eliminated by the following means: Conflict (3) involves conflicts between operations that modify the header word and the *store* operation, which modifies other page elements. This conflict can be eliminated by providing *fetch* and *store* operations on pages that allow each element of the page to be read or written individually, without interfering with other elements of the page. Conflict (1), interaction between vector *fetch* and *store*, can be eliminated by making the page *fetch* and *store* operations atomic. Conflict (2) involves the vector *store* operation, which may have to decrement the reference count of the object whose reference was overwritten. To make sure that the correct reference count is decremented, we simply have the page *store* operation (which is atomic) return the overwritten reference. Conflicts (4) and (5) involve

Figure 20. Synchronization analysis of vector operations.

	Create	Equal	Fetch	Store	Size	Incr_rc	Decr_rc	Reclaim	Mark	Unmark	Marked
Create	A	-	-	-	-	-	-	-	-	-	-
Equal	A	B	-	-	-	-	-	-	-	-	-
Fetch	A	B	B	-	-	-	-	-	-	-	-
Store	A	C	1	2	-	-	-	-	-	-	-
Size	A	B	B	C	B	-	-	-	-	-	-
Incr_rc	A	C	C	3	C	5	-	-	-	-	-
Decr_rc	A	C	C	3	C	5	5	-	-	-	-
Reclaim	A	A	A	A	A	A	A	A	-	-	-
Mark	A	C	C	3	C	4	4	A	A	-	-
Unmark	A	C	C	3	C	4	4	A	A	A	-
Marked	A	B	B	C	B	C	C	A	A	A	A

A: no conflict; concurrent operations involving any particular vector are not possible
B: no conflict; both operations are reads
C: no conflict; no intersection between bits read and bits written

1: read/write conflict; read must get either old or new value
2: stores must be performed in sequence; each store must obtain the old (overwritten) value so that its reference count can be decremented
3, 4: the operations write disjoint bits; synchronization is needed only if the writes are implemented as updates, e.g., updating a whole word to write a single bit
5: update conflict; both operations write a new value based on the old value; operations must be performed in sequence

---

operations that modify parts of the header word. These conflicts can be eliminated by providing atomic page operations to perform the appropriate manipulations on the header word (element 0 of the page).

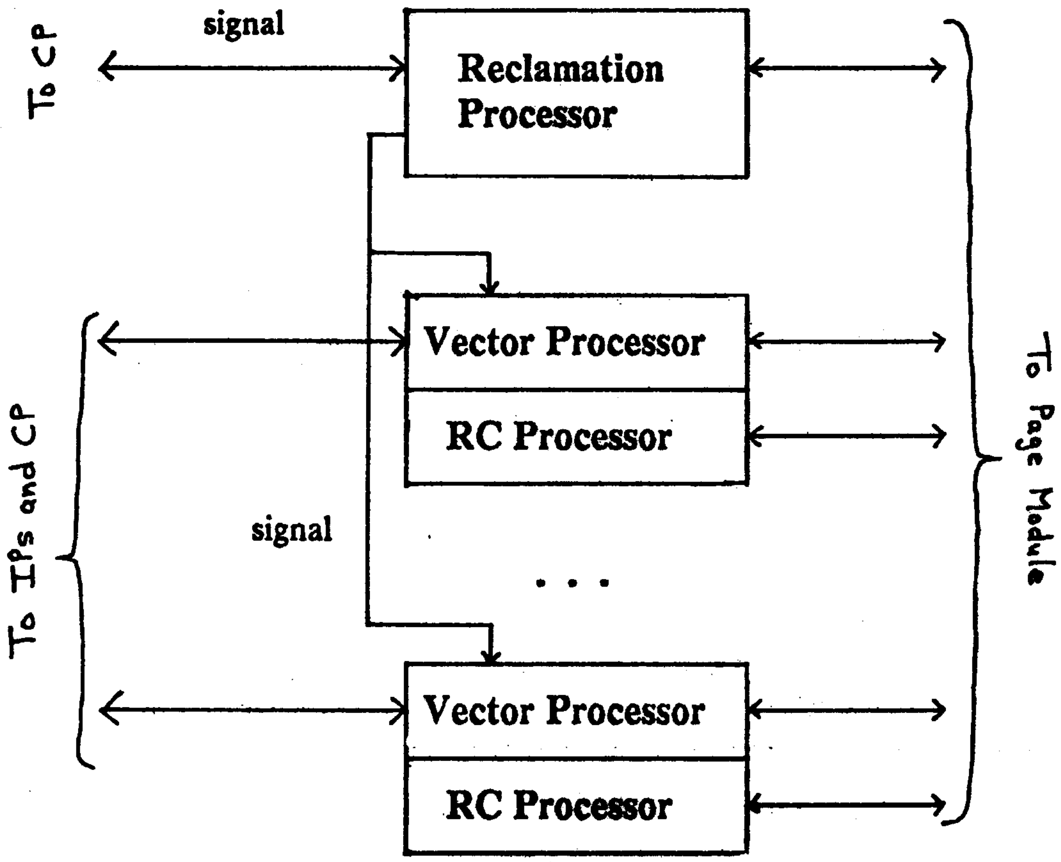
What we have just done is to push all of the vector synchronization problems onto the page module, which now must support a number of atomic operations of its own. As a result, we eliminate the need to have any explicit synchronization in the vector module itself. While adding reference count and mark bit operations to the page module is somewhat distasteful, the elimination of a second level of synchronization simplifies the design considerably. As an aside, although we will name these new page operations after their vector counterparts (i.e., *incr_rc*, *decr_rc*, *mark*, and *unmark*), their actual specification would not be in terms of reference counts and mark bits (since pages don't have reference counts or mark bits), but in terms of manipulating the contents of the bstring object that is the contents of element 0 (the vector header word).

8.3 Vector Module Design

A block diagram of the vector module is shown if Figure 21. The connection to the processing module consists of a number of bidirectional ports, one for each instruction processor and one for the control processor. Each processor sends requests to the vector module and receives replies via its associated port. The requests sent to the vector module correspond to invocations of the external vector operations. A processor sends a single request at a time, always waiting for the reply before sending another request. (The rationale for this arrangement is presented in Section 4.4.)

On the other side are a number of bidirectional ports connecting the vector module to the page module. These ports are used by the vector module to send requests to the page module and receive the corresponding replies. Again, each

Figure 21. Vector module block diagram.



port supports only one outstanding request at a time. Thus, the number of ports corresponds to the maximum number of concurrent requests being processed by the page module.

The vector module itself consists of a number of *vector processors*, plus a *reclamation processor*. Each vector processor listens to a single input port. Its function is to perform each incoming request by submitting one or more requests to the page module through its output port. The function of the reclamation processor is to process the GQs and to perform garbage collection. These two activities are never performed at the same time.

Associated with each vector processor is a *reference count processor*, whose function is to perform any reference count operations resulting from the processing of vector operations. Each reference count processor has its own port to the page module, allowing it to perform page operations in parallel with the vector processor. This concurrency is important, as the execution of a *store* request may generate reference count operations on two other vectors (the vector whose reference is being stored and the vector whose reference is overwritten). There is no need to hold up the reply to the original request while waiting for the reference count operations to be performed (which might involve waiting for the affected vectors to be swapped into primary storage). The page module can be designed to always select a request from a vector processor before selecting a request from the corresponding reference count processor, if both ports contain requests. The idea is to minimize the interference with vector operations caused by reference count operations.

The reference count processor contains an internal queue to hold reference count operations waiting to be performed. Note that because the reference count operations resulting from a *store* operation are not performed as part of a single, atomic operation, reference count operations generated by different vector processors can be performed out of order. As described in Section 6.7, reference count operations in our system do not have to be performed in the same order as the generating *store* operations.

In addition, each reference count processor maintains its own queue of GQ entries. As described in Section 6.7, only the total number of GQ entries for each object is important, not the order in which the entries are processed. Thus, each reference count processor can collect GQ entries on its own separate queue. At the end of the GQ cycle, all of the separate queues can be passed to the reclamation processor to be processed together. Having a separate GQ for each vector processor rather than a single GQ for all vector processors eliminates the need to synchronize access to a shared data base and allows GQ operations from different vector processors to be performed concurrently (to the extent permitted by the page module).

The various GQs will be implemented as lists of *pages*, linked together by their first elements. The queue entries will simply be vector references; note, however, that these references are *not* counted in the reference counts! Unused elements in these pages can be initialized to some distinguished *undefined* value. A reference count processor will obtain new queue pages from the page module as needed and chain them into the list. Using shared memory to implement the GQs eliminates the need to have a separate intermodule communication mechanism.

The decision to begin a new GQ cycle is made by the reclamation processor. A new GQ cycle cannot begin until the previous GQ has been processed by the reclamation processor. The reclamation processor will begin a new GQ cycle sometime after it finishes processing the previous GQ.

The procedure for establishing quiescence and beginning a new GQ cycle is as follows: The reclamation processor notifies the control processor of its intention to establish quiescence via a special asynchronous control line, labeled "signal" in Figure 21. Upon receipt of this signal, the control processor will cause all references in the processing module to be stored in the memory module. After all the references have been stored and replies have been received for all requests sent to the memory module, the control processor will notify the the reclamation processor that it is done via the same control line. At this point, no more operation requests will be received by the vector processors.

The reclamation processor then notifies the vector processors that quiescence is being established, using another asynchronous control line. After pending requests have been fully processed (including all reference count operations) and replies have been received for all requests sent to the page module, each vector processor will notify the reclamation processor by storing a reference to its GQ (a reference to the first page in the list) in a special *GQ table page* (GQTP). The elements of the GQTP are initially set to some distinguished *undefined* value. The reclamation processor can determine that the vector processors are ready by waiting for them to store GQ references in the GQTP. (The reclamation processor is busy-waiting, which seems acceptable in this situation.)

After all activity ceases in the vector processors and the reference count processors, the reclamation processor can obtain the various GQs and begin to process them. The reclamation processor must wait until all activity ceases to be sure that the GQs have reached their final states. After the reclamation processor obtains the GQs, it can reset the GQTP elements to *undefined*. It can then notify the control processor to resume operation, using the asynchronous control line. Each vector processor, when it receives its next request from the processing module, will allocate a new GQ in which subsequent GQ entries will be stored.

8.4 Page Module Specification

The function of the page module is to implement a number of primitive operations. The page module receives requests from the vector module on a number of bidirectional ports. Each request corresponds to an invocation of one of the page operations. For most requests, the page module will eventually deliver a reply to the corresponding port. No additional requests can be submitted on that port until that reply has been delivered.

The page operations are listed in Figure 22. Most of these operations have an obvious correspondence with the vector operations previously described. One difference is that page sizes will be identified by size numbers (zone numbers) rather than the actual size. Two new operations are *sweep_reset* and *sweep_next*. These operations are used by the reclamation processor to perform the sweep phase of the garbage collection. The *sweep_reset* operation is used to reset an internal counter in the page module at the beginning of the sweep. Then, *sweep_next* is called repeatedly; on each call it returns a reference to the "next" page in the virtual memory. (The order can be chosen to optimize the performance of the sweep phase.) Both allocated and free pages will be returned. The reclamation processor can determine whether a page is allocated or free by examining the first element of the page; we assume that header words and free list pointers can be distinguished from each other and from anything else using some type code bits.

All of the page operations are atomic. Any set of page operations processed concurrently by the page module must produce a result equivalent to performing the operations in some order.

Not all possible sequences of page operations are valid. In particular, once a given page "object" has been deallocated, it is improper to perform additional operations on it. Of course, eventually the same page "object" will be reallocated, after which time operations on that page may again legally be performed. Because the page module cannot check the intent of a request (was it intended to be performed on the "current" use of a given page or one that was previously deallocated?), there is no way for the page module to detect all invalid

Figure 22. Page module operations.

```
alloc = proc (size_number: bstring) returns (page) signals (no_storage)
dealloc = proc (p: page)  % No Reply
equal = proc (p1, p2: page) returns (bstring)
size = proc (p: page) returns (bstring)
fetch = proc (p: page, index: bstring) returns (any) signals (bounds)
store = proc (p: page, index: bstring, element: any) signals (bounds)
incr_rc = proc (p: page) returns (bstring) % Returns old value of reference count
decr_rc = proc (p: page) returns (bstring) % Returns new value of reference count
mark = proc (p: page)
unmark = proc (p: page)
touch = proc (p: page) returns (bstring)
sweep_reset = proc ()
sweep_next = proc () returns (page) signals (no_more)
```

sequences of operation requests. Instead, we must simply *require* that all sequences of page operations be valid; where multiple operations are submitted concurrently, we must require that *any* ordering of those operations be a valid sequence. This requirement is simply a condition of correctness of the system; if this requirement is not satisfied, the system cannot be considered correct.

There is really only one new restriction imposed by this requirement: No page may be deallocated if there are pending operations on that page. (The *touch* operation is the only operation that may initiate a swapin without waiting for it to complete. For convenience, we will consider a *touch* operation to be pending until some other operation (e.g., *fetch*) is performed on the page.) This restriction is interesting because it means that the page module does not have to be prepared for conflicts between deallocation and other operations; it does not have to worry about someone deallocating a page while it is being swapped in. The restriction is easy to satisfy: Only the reclamation processor performs deallocation; it does so only when the page is inaccessible to all other modules. Thus, there should be no possibility of concurrent operations on a page being deallocated.

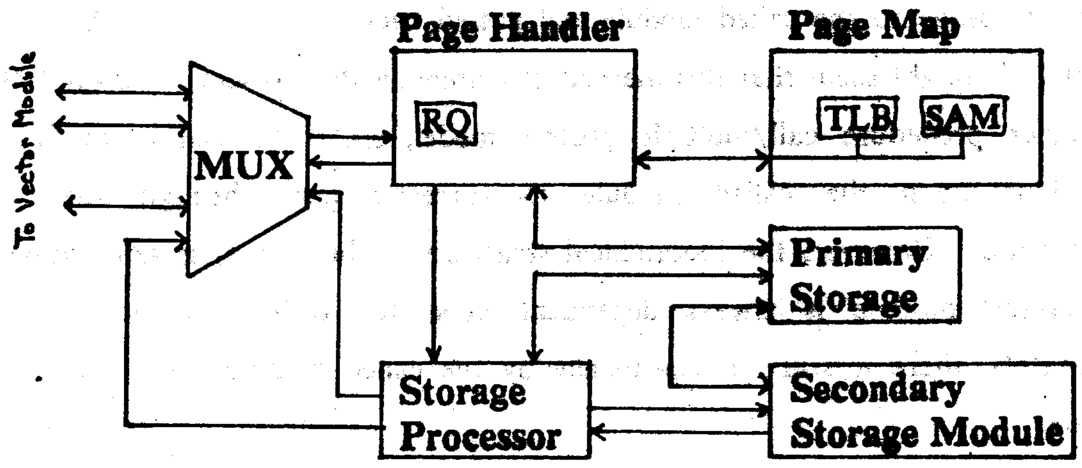
8.5 Page Module Design

This section describes the design of the page module. We begin by presenting a block diagram and describing the functions of the various internal modules. Next, we describe the operation of the page module by considering how it processes the various kinds of input requests. Finally, we discuss a number of specific issues, including page replacement, synchronization, and flow control.

8.5.1 Page Module Organization

A block diagram of the page module is presented in Figure 23. The page module consists of a number of modules that communicate via messages sent over unidirectional or bidirectional channels. The primary storage module and the

Figure 23. Page module block diagram.



page map both behave as subroutines: a request is accepted and a reply is quickly returned over the same channel. Other modules have separate input and output ports and use a continuation model of communication [17]. In this model, each message (implicitly or explicitly) carries with it an indication of where the result of the operation is to be sent and what is to be done thereafter. For example, when the storage processor sends a request to the secondary storage module, it does not wait for a reply. Instead, when the secondary storage module finishes processing the request, it will send another message to the storage processor. This message will be treated as a new request by the storage processor; the message will contain all the information needed to instruct the storage processor about what to do with the message.

The continuation model of communication is especially suitable for implementing concurrent operations in hardware systems without a global state. It removes the need for a calling module (e.g., the storage processor) to maintain information about the state of the transaction after sending a request to a called module (the secondary storage module). It removes the need for the calling module to match that state information with the reply when the reply is received from the called module. Instead, the state of the transaction is passed with the request message to the called module and passed back in the reply to the calling module. It might seem that this method of communication is less structured than the strict procedure call/multiple process model, since the calling module is dependent upon the called module to correctly pass through the state information. However, the hierarchical structure of the system is not damaged: the calling module is always dependent upon the called module (e.g., to terminate!); there is still no dependency by the called module upon the calling module.

8.5.2 Module Functions

In this section, we will briefly describe the functions of each of the internal modules of the page module. The MUX module arbitrates requests arriving on the input ports. Whenever the page handler is ready to receive a new request, the MUX will select a waiting request from one of the input ports. Each request will include a port identification (supplied by the originating module), which will later be used to address the reply to the proper input port.

Replies may be generated by either the page handler or the storage processor. Each reply contains a port address. When either of these modules sends a reply message to the MUX, the message will be delivered to the specified port. Because each port may submit only one request at a time, the designated port will always be ready to receive the reply message.

There is one additional input to the MUX, which is connected to an output port of the storage processor. This channel is used by the storage processor to send messages to the page handler. It should have a higher priority than the input ports. The MUX thus serves as the only arbiter of incoming messages to the page handler.

The page handler processes input requests in sequence. It has exclusive access to the page map, which it uses to determine the primary storage addresses of pages being operated upon. In most cases, the page handler can quickly process a request and send a reply back through the MUX. For slow operations, such as *alloc* and *dealloc*, the page handler will pass the request to the storage processor for handling.

If an operation is performed on a page that is not in primary storage, the page handler will queue the operation on an internal request queue (RQ) and send a *swap_in* request to the storage processor. When the swapin has completed, the storage processor will notify the page handler and the page handler can perform the queued requests for that page. Queuing requests on the RQ is contrary to the continuation model described in the previous section. However, the alternative of sending the request along with the *swap_in* request to the storage processor is unacceptable, since additional requests received by the page handler for a page being swapped in do not cause *swap_in* requests to be sent to the storage processor.

The primary function of the storage processor is to perform primary storage allocation and deallocation. In this capacity, the storage processor also serves to initiate page replacement when the amount of free primary storage becomes too low. In addition, the storage processor serves to direct traffic between the page handler and the secondary storage module. Finally, the storage processor maintains the counters needed to implement the *sweep_reset* and the *sweep_next* operations.

The page map maintains a mapping between the secondary storage address and the primary storage address of each page in primary storage. As discussed in Chapter 5, the page map is organized as a large set associative memory (SAM), along with a small, fast translation lookaside buffer (TLB). The TLB will use store through, so that when a new entry is to be added to the TLB, an existing entry can simply be discarded.

The secondary storage module performs secondary storage allocation and data transfers between primary and secondary storage. It processes requests submitted by the storage processor and sends replies back to the storage processor. It is assumed to be capable of performing a number of operations in parallel to reduce the average secondary storage access time.

The primary storage module is a conventional addressable memory supporting read and write operations. Requests are submitted by the page handler, the storage processor, and the secondary storage module.

8.5.3 Page Module Operation

In this section, we informally describe the operation of the page module as it performs each of the various page operations. A more complete, although still informal, description of the operation of the various modules is given in Appendix II.

All of the operations *fetch*, *store*, *incr_rc*, *decr_rc*, *mark*, and *unmark* are processed in basically the same way. For example, when the page handler receives a *fetch* request, it will first look up the page in the page map. If there is no entry for the page, then the page must be swapped in. The page handler will enter a new entry in the page map. This entry will indicate that the page is being swapped in, to prevent subsequent requests for the same page that arrive before the swapin completes from initiating a second swapin. The page handler will also send a *swap_in* request to the storage processor, asking that the page be swapped in. In addition, the page handler will queue the original request on the RQ, to be processed again after the page is swapped in. (If the RQ is full, the page handler can *reject* the request by sending a rejection reply message to the requesting module, informing it that the request should be resubmitted at a later time.)

If there is an entry for the page, but it indicates that a swapin is in progress, then the request is simply queued on the RQ. Otherwise, the page handler obtains the primary storage address of the page from the page map entry, performs the *fetch* operation, and sends the result back to the requesting module.

When the storage processor receives a *swap_in* request, it first sends a message to the secondary storage module to find out what the size of the page is. (Recall that the size of a page can be determined by its secondary storage address, using a table that maps block addresses to zone numbers.) When the reply is received, the storage processor then allocates a primary storage page of the desired size and sends a *swap_in* request to the secondary storage module. The secondary storage module will transfer the page and send a *swap_in_done* message to the storage processor, which forwards it to the page handler. The page handler updates the page map entry to include the primary storage address and then performs any queued requests for that page.

There is one additional complexity in this description. Whenever a new entry is added to the page map, it is possible that the corresponding set of the set associative memory will be full, in which case an existing entry in that set must be pushed out of the associative memory (and out of primary storage). (For synchronization reasons, we require that entries for pages in transit not be forced out of the page map. Should the set be full of in-transit entries, which is unlikely, the original request must be rejected. The requesting module would then have to resubmit the request at a later time.) Whenever an entry is forced out of the page map, the page handler will send the page map entry to the storage processor in a *swap_out* request.

Pages must also occasionally be removed from primary storage to make room for other pages to be swapped into primary storage. The decision to remove a page from primary storage is made by the storage processor, as will be described in the next section. The storage processor sends a *remove* request to the page handler, which removes the page from the page map and sends the page map entry in a *swap_out* request back to the storage processor.

Upon receipt of a *swap_out* request, the storage processor first checks the attached page map entry to see if the primary storage copy of the page has been modified. If not, all that needs to be done is to deallocate the primary storage page. Otherwise, the primary storage page must be written to secondary storage. In this case, the storage processor forwards the *swap_out* request to the secondary storage module. The secondary storage module will write out the page and then send a *swap_out_done* message back to the storage processor. The storage processor will then deallocate the primary storage page.

When an *alloc* request is received by the page handler, it simply passes that request to the storage processor. The storage processor in turn passes the request to the secondary storage module. The secondary storage module will allocate a secondary storage page and pass its address back to the storage processor. Upon receipt of this message, the storage processor will allocate a primary storage page, initialize it, and send both the primary and secondary storage addresses to the page handler. The page handler will enter this information in the page map and send the secondary storage address back to the requesting module.

When a *dealloc* request is received by the page handler, it looks up the page in the page map. If there is an entry, it is removed, and the primary storage address is sent along with the secondary storage address to the storage processor. Otherwise, only the secondary storage address is sent. The storage processor will pass the *dealloc* message to the secondary storage module, which will deallocate the secondary storage page. (No reply is returned to the storage processor.) At the same time, the storage processor will deallocate the primary storage page, if any. No reply is returned to the page handler or the requesting module.

The *equal* operation is handled solely by the page handler, as all that is involved is comparing the two references. The *size* operation, on the other hand, is passed via the storage processor to the secondary storage module, which determines the page size using its block map. The reply is sent via the storage processor to the requesting module.

The *touch* operation is handled similarly to the *fetch* operation, except that if a swap-in is required, no request is queued. In all cases, an immediate reply is sent indicating whether or not the page is in primary storage.

Both the *sweep_reset* and *sweep_next* operations are passed directly to the storage processor, which replies directly to the requesting module.

8.5.4 Page Replacement

The function of page replacement is to select pages to be removed from primary storage to make room for other pages to be brought into primary storage. In this section we present a simple page replacement method that is easily implemented.

As described above, when a set of the set associative memory (SAM) becomes full and a new entry must be added to the set, some existing entry in that set must be forced out of the page map. Thus, the set associative memory requires a replacement algorithm of its own. The least recently used (LRU) algorithm is a likely choice for the set associative memory replacement algorithm, because it is easily implemented and gives good performance.

LRU replacement can be implemented as follows: Each entry in the SAM can contain the *stack position* of the entry in the LRU stack [26]. When a page is accessed, the stack positions of the entries in the set are adjusted so that the accessed page has stack position 1 (the top of the stack), and all entries previously higher in the stack than the accessed page are moved down one (their stack positions are increased by one). All of these operations are performed in parallel. To find the least recently used entry, one simply searches for an entry whose stack position equals the number of entries in the set. (If the least recently used entry were marked as in-transit, then the next least recently used entry would have to be found, etc. It is unlikely, although not impossible, for all the other entries in the set to have been accessed between the time that a swapin was initiated and the time the swapin completes.)

Our simple page replacement algorithm is based on that implemented by the set associative memory. Whenever we wish to select a page for replacement, we choose a particular set (e.g., by keeping a counter that cycles through all set numbers) and then force the least recently used not-in-transit member of that set (if any) out of the associative memory (and therefore, out of primary storage). The only difference between this action and the normal LRU action of the set associative memory is that in this case the set may not be full (although it is likely to be nearly full). Thus, the least recently used entry may be empty, forcing the search to continue with the next least recently used entry, as described above. Page replacement continues as long as the amount of available primary storage is below some desired level.

The decision to cause a page to be removed from primary storage is made by the storage processor, which is in charge of primary storage allocation. The storage processor will send a *remove* message containing the selected set number to the page handler. Upon receipt of the *remove* request, the page handler will request the page map to remove the least recently used not in-transit page in the specified set. The page handler then sends a *swap_out* message to the storage processor.

Our page replacement algorithm is an approximation of the LRU algorithm. It differs from the LRU algorithm only in the relative ordering of pages in different sets of the SAM. The obvious disadvantage of an LRU page replacement algorithm in our system is that the sizes of pages are not taken into account. Suppose a free primary storage page of a particular size is needed. Our algorithm may remove a number of pages of other sizes before a page of the desired size is removed. These unnecessary page removals could later result in extra secondary storage accesses. On the other hand, if the block allocation scheme described in Section 7.3 is used for primary storage allocation, then it may occasionally be desirable to remove *particular* pages of other sizes to create empty blocks that can be reassigned to the desired zone. Thus, while the proposed page replacement method will work, it is clearly not optimal.

We have not attempted to find the optimal page replacement algorithm for our system. However, it should be noted that cleverer schemes may require additional hardware or may take more time to perform. For example, the ability to remove a "not recently used" page of a particular size probably requires that each page map entry contain the zone number of the page. The ability to remove specific pages to create free blocks requires the ability to find the page map entry of a page given its *primary storage* address, which probably requires an additional data base mapping primary storage addresses to secondary storage addresses. When evaluating other page replacement algorithms, the cost of implementation must be balanced against the assumed benefit of a reduced secondary storage access rate.

8.5.5 Synchronization

The simplest method for implementing atomic operations is to have all operations performed by a single module that performs only one operation at a time, in sequence. We have tried to use this method to the greatest extent possible. The page handler does process only one request at a time. In most cases it will completely process the request and send a reply to the requesting module before reading the next request. Using this method, there is only one point of synchronization, the MUX. Because the page handler has exclusive access to the page map, no synchronization is needed to control access to the page map.

Unfortunately, strictly following a one request at a time discipline would lead to poor performance. Therefore, some concurrency is allowed. Allocation and deallocation are performed concurrently with other operations. Swapins and swapouts are performed concurrently with other operations. These concurrent operations require additional synchronization.

The various allocation data bases are synchronized by making them private resources of specific modules. The storage processor has exclusive access to the primary storage allocation data base; the secondary storage module has exclusive access to the secondary storage allocation data base. Each module can ensure consistent access to its own data base.

Other synchronization problems involve the coordination of swapins and swapouts with other activities. Swapins are coordinated with other events using the traditional device of an in-transit page map entry. When a swapin is first requested, an entry is added to the page map indicating that the page is in transit. This entry is later modified when the swapin completes. Input requests arriving in the meantime can tell from the in-transit entry that a swapin is in progress and avoid starting a second swapin. *Replace* requests similarly will avoid selecting this page to be swapped out. Because each of these state changes are performed as atomic operations by the page handler, no race conditions are possible.

The synchronization of swapouts is a bit more difficult. There are two problems that must be avoided: First, if a swapout is immediately followed by a swapin, the swapin must not read the old secondary storage copy of the page. Second, if a swapout is immediately followed by a deallocation, then the swapout and the deallocation must not interfere. One possible solution to these problems is to use state information in the page map, as was done to coordinate swapins. However, this solution is not acceptable. When a swapin is requested, a new entry must be added to the page map. Adding this new entry may force out an old entry, initiating a swapout. If the old entry had to stay in the page map until the swapout were completed, then the swapin could not begin until the swapout had finished. Meanwhile, the page handler would not be able to process any new requests.

While various methods could be used to make this solution work, there is a better solution. The interference between swapouts and other events can be eliminated simply by ensuring that all secondary storage transfers for a given secondary storage page are performed in the order that they were initiated by the page handler. For example, we know that the page handler will not generate a swapout until after the previous swapin has completed. We know that the page handler will not generate another swapin until after it has sent out the swapout request. If the secondary storage transfers implied by these requests are performed in that order, then the second swapin will obtain the correct data.

Similarly, we know that after the page handler generates a *dealloc* request for a page, no further requests will be generated for that page until after the page is reallocated. Thus, any *swap_out* request must have been issued before the *dealloc* request. If the secondary storage transfers corresponding to the *swap_out*, *dealloc*, and *alloc* requests are performed in that order, then first the page will be written by the *swap_out*, then it will be written by the *dealloc* (which writes the old free pointer in the first element of the page), and finally it will be read by the *alloc* (which reads the free pointer). It is possible that the primary storage page may be deallocated before the *swap_out* is completed, in which case the *swap_out* transfer will write garbage into the secondary storage page. However, the immediately following *dealloc* transfer will write meaningful data.

8.5.6 Flow Control

The page module consists of a number of modules that send messages to each other. Most of these modules can handle a number of concurrent requests. However, each module can store only a limited number of messages at any one time. Should this limit be reached, it could not accept further input messages until the processing of some of the stored message had been completed.

If a module R is not accepting input messages, then another module S that wants to send a message to R would not be able to. It must either hang until R is ready to accept the message (until which time S will not accept additional messages), or it must queue messages for R internally until R is ready to accept them. If S queues messages for R, it can eventually run out of internal storage, in which case it again must stop accepting new input messages.

Thus we see that if one module stops accepting messages, another module that sends messages to it may also be forced to stop accepting messages as a direct result. If the system contains directed cycles, then deadlock may result, as all of the modules in a cycle may be stopped waiting for another one to accept messages.

In our design, there are three directed cycles (not counting the subroutine-like connections to the page map and the primary storage module). We must check each of these cycles to be sure that deadlock can be avoided.

One cycle (actually a set of cycles) consists of an input port (connected to a processor in the vector module) and the page handler. (In some cases, the storage processor is also involved.) This cycle is no problem because each requesting processor can submit only one request at a time on a given port. For those requests that generate replies (all generate replies except *dealloc*), the requesting processor is required to wait for the reply before sending another request on that port. We are thus guaranteed that any reply sent to an input port will be accepted.

Another cycle consists of the storage processor and the secondary storage module. All messages sent from the secondary storage module to the storage processor are in response to messages sent from the storage processor to the secondary storage module. Therefore, the storage processor can predict the amount of storage needed to store anticipated replies from the secondary storage module. It can avoid sending a message to the secondary storage module unless storage has been reserved for the reply. This requirement can be reflected to the input from the page handler: the storage processor can refuse to accept messages from the page handler unless it can reserve enough space to completely process the request, including storage to hold any replies to requests sent to the secondary storage module.

The third cycle is the most complex. It consists of the page handler and the storage processor. Most of the interactions between the page handler and the storage processor are initiated by the page handler. The *swap_out* and *dealloc* requests generate no replies by the storage processor. The *alloc* and *swap_in* requests eventually cause reply messages to be sent from the storage processor to the page handler. Other requests generate replies from the storage processor directly to the requesting input port.

The storage processor initiates only one kind of interaction with the page handler, using the *remove* request. The page handler replies with a *swap_out* request. This message sequence is used to cause pages to be swapped out to make room in primary storage for other pages. It may be necessary to swap out a number of pages before the storage processor can satisfy *alloc* or *swap_in* requests.

To avoid deadlock, we must ensure that if the storage processor ever stops accepting messages from the page handler, then there is at least one request being processed by the storage processor whose processing can be fully completed without requiring the page handler to accept additional messages. When this message is fully processed, storage will be made available to allow a new message to be accepted by the storage processor. (For simplicity, we assume that all messages require an equal amount of internal storage.) By our assumption, this new message also can be fully processed without requiring the page handler to accept additional messages. Included in this set of messages are replies to previous requests from the storage processor to the page handler; such reply messages allow pending requests to be completed by the storage processor, thus reducing the number of requests pending in the storage processor.

This requirement has two implications: One implication is that the storage processor must be able to process other requests at the same time it is attempting to send a message to an unresponsive page handler. A multi-process storage processor is thus necessary, as is the separate storage processor port for sending messages to the page handler. (Note, however, that this multi-process implementation must not reorder requests on their way to the secondary storage module!)

The second implication is that the number of concurrent requests that may require the storage processor to send a message to the page handler must be bounded. The requests that require messages to be sent from the storage processor to the page handler are *alloc* and *swap_in*. Both messages require a reply to be sent from the storage processor to the page handler; both may require *remove* requests to be sent to the page handler.

The number of concurrent *alloc* requests is bounded by the number of input ports, which are the ultimate source of the requests. *Swap_in* requests, however, are not bounded by the number of input ports, as a sequence of input *touch* requests could generate virtually an unlimited number of concurrent *swap_in* requests. To control the number of *swap_in* requests we must limit the number of *touch* requests that can be submitted before performing a real operation on a touched page (which would force the requestor to wait until the page had been swapped in). The *touch* operation is used only by the garbage collector; there is at most one pending swapin per garbage collector "process" (see Figure 15). Thus, by restricting the number of garbage collector processes, we can limit the number of concurrent requests produced by *touch* operations.

In summary, the number of concurrent *alloc* and *swap_in* requests is bounded by the number of input ports plus the number of garbage collector processes. These numbers can be fixed (maximum values chosen) by the system designer. Avoiding deadlock is thus simply a matter of providing enough storage in the storage processor to allow the maximum number of such requests to be processed concurrently, plus room for at least one other request. (For safety, the page handler can keep a count of the number of *alloc* and *swap_in* requests being processed by the storage processor. It can reject input messages if the count gets too high.)

8.6 Improvements

A number of improvements can be made to the design presented in this chapter to increase the performance of the system. In particular, there are a number of ways that the system can be "tuned" to provide greater throughput and to balance the capacities of the various parts of the system. For example, because all requests pass through the page handler, it is likely that the page handler would be a bottleneck, limiting the throughput of the memory module. However, the throughput of the memory module can be increased simply by providing more than one page handler (each with its own associated page map). The technique is similar to conventional interleaving. The virtual address space is divided into two or more subspaces, probably based on the same hashing function used to compute the SAM set numbers. Each page handler handles pages in one of the subspaces. Page operations directed at a particular page will be sent to the corresponding page handler; *alloc* requests can be sent to any page handler, as they are simply passed to the (single) storage processor. To achieve the maximum performance improvement, the primary storage module should probably also be interleaved in the conventional sense to provide sufficient throughput for multiple page handlers. It is also fairly easy to provide multiple secondary storage modules if greater secondary storage bandwidth is needed.

Another plausible improvement would be to cache the first elements of pages in the TLB. The first elements of most pages are vector header words. These elements are accessed frequently to perform bounds checking and update reference counts. One could also provide additional page module ports for use by the reclamation processor, so that it could perform a number of reference count operations in parallel.

In summary, the design is quite flexible in terms of permitting adjustments to achieve better, more balanced performance. Exactly what changes should be made would best be determined after simulation studies or perhaps the construction and measurement of a prototype.

9. Conclusions

This thesis has presented the design of a computer system that directly supports an object-oriented machine language. The machine provides a single, large universe of objects shared by multiple processes.

The universe of objects is implemented using a multi-level memory system. Each object is represented by a single "page"; the system supports a number of different page sizes. Objects (pages) are identified by their secondary storage addresses and are transferred individually between primary and secondary storage. A large set associative memory maps from the secondary storage addresses of objects in primary storage to their primary storage addresses. Storage is allocated from a number of zones; each zone provides pages of a single size and contains its own list of free pages. Physical storage is divided into fixed-size blocks; each block is (statically or dynamically) assigned to a single zone. Automatic storage reclamation is implemented using queued reference counts and occasional garbage collection.

An implementation of the system was described in terms of a number of specialized processor modules communicating via messages. Multiple processors are used to improve performance and to achieve a more modular system structure.

9.1 Evaluation

The major contribution of this thesis is a new design for a computer system that supports a single, large address space of objects. The proposed design has a number of advantages, and some disadvantages, in comparison with other designs providing similar capabilities:

The first advantage is that the machine supports a uniform address space. There is no concept of "areas" or other groups of objects needed to allow adequate performance. While we do not claim that object grouping is inherently evil, or that it could never be useful, any system that requires programmers or users to think about grouping objects will be more complex and more difficult to use than one that doesn't. The corresponding disadvantage of our design is that its performance is more severely limited by the secondary storage access time. While it is difficult to predict how short the access time must be (among other things, it depends upon how the system is used), it is clear that a system that swaps groups of objects can achieve better performance with slower secondary storage devices.

Another advantage of the proposed design is that it performs incremental automatic storage reclamation, using reference counts. Storage reclamation is performed continuously, without requiring frequent or unpredictable interruptions of service. However, periodic garbage collection is still required, which (in our proposal) requires the the system be stopped for short periods of time at scheduled intervals. A disadvantage of our design is that the need for garbage collection depends upon program behavior (the rate of generation of cyclic garbage).

Another advantage is that (we believe) the proposed design is capable of good performance. The virtual memory mapping is performed efficiently by a hardware set associative memory. We have shown how a large set associative memory can be constructed using a minimal amount of special-purpose hardware. In the proposed design, memory management activities, such as allocation and swapping, are performed concurrently with other operations. The secondary storage allocation algorithm limits the number of needed secondary storage accesses to at most one per allocation and deallocation; many of these accesses are easily eliminated by maintaining portions of the free lists in fast storage. The use of queued reference counts reduces the time overhead of automatic storage reclamation and allows most reclamation processing to be performed concurrently with normal operations. References are compact, e.g., 32 bits.

A disadvantage of using a set associative page map is that restrictions are placed on the possible collections of objects that can simultaneously reside in primary storage. On average, these restrictions will have little effect; however, the potential exists for degradation of paging performance.

Another advantage of the design is that it can be used in a multiprocessor configuration (multiple IPs). The ability to use multiple processors allows the processing power of the system to be adjusted over a wider range to support the computations performed on the data stored in the virtual memory. The design is flexible in that the machine can be configured (by the duplication of various modules) to provide greater throughput and to balance the throughputs of the various components.

Finally, and significantly, although multiple processors are being used and automatic storage reclamation is being performed in parallel with normal computation, the basic concepts of the system are relatively simple. Using queued reference counts, the set of events that cause reference count operations is small, well-defined, and localized to the memory module. The notion of quiescence is easy to understand and verification of its correct implementation should be straightforward. Furthermore, the reference count implementation avoids many synchronization problems.

Three disadvantages of the proposed design have already been mentioned: the need for fast-access secondary storage devices, the need for occasional garbage collection, and the potential for degraded paging performance. The other disadvantage of the design is that it entails a higher hardware cost, compared to other designs. Additional hardware is required for the set associative memory, the many hardware modules (processors), and the required module interconnections. Other costs result from underutilization of primary and secondary storage:

Primary storage utilization is reduced for a number of reasons. The biggest factor is the set associative memory, which is in effect constructed out of primary storage. The appropriate size of the set associative memory depends upon the expected average object size; the number of associative memory entries should approximate the expected number of objects in primary storage. If the average object size is ten words (nine elements per vector), then the set associative memory should be one-fifth the size of the actual primary storage (assuming each associative memory entry occupies two words), for an overhead of 16%. If the average object size is only four words (three elements per vector), the overhead is 33%. Additional underutilization is caused by fragmentation. Internal fragmentation occurs because extra storage is allocated to objects whose sizes are different than any of the supported page sizes; internal fragmentation can be limited to 5-15% by choosing an appropriate set of supported page sizes. External fragmentation occurs because blocks of storage are dedicated to providing particular page sizes; the amount of storage wasted because of external fragmentation is difficult to predict. Additional primary storage is occupied by the GQs (about 6K words) and by objects on the GQs waiting to be reclaimed (15-50K words, roughly). The actual amounts depend upon the rate of garbage generation and the GQ cycle time; to some extent, one can trade off storage for time by changing the GQ cycle time.

On the other hand, there are some factors that improve primary storage utilization. First, the swapping of individual objects makes more effective use of primary storage. Second, using reference counts, garbage will be reclaimed sooner than in a system using garbage collection (although a traditional reference count implementation would reclaim garbage even sooner). These effects could cancel out some of the effects listed above. The main remaining factor is probably the set associative memory. Although 33% underutilization of primary storage is probably more than in conventional systems, it is not overwhelming.

Reduced secondary storage utilization is caused by three factors: internal fragmentation, external fragmentation, and accumulating cyclic garbage. As in the case of primary storage, internal fragmentation can be limited to the range of 5-15% by proper selection of the supported page sizes. External fragmentation and cyclic garbage are more difficult to predict, as they are dependent upon program behavior. Basically, extra secondary storage must be provided to allow for changing object size distributions and for generated cyclic garbage. The amount of extra storage needed depends upon program behavior and the desired rate of garbage collection, but not on the total secondary storage size. Thus, for large secondary storage sizes, the fraction of wasted storage should be low.

Another contribution of this thesis is that it demonstrates how multiple processors can be used to simplify the structure and improve the performance of a system that supports multiple processes and a large virtual memory. We use the term processor here to include all of the major active hardware modules, not just the instruction processors.

The system is constructed hierarchically out of modules that perform well-defined functions. At the top-most level, the system is divided into two major modules, the processing module and the memory module. The processing module interprets procedures and implements multiple processes. It consists of a number of instruction processors, which interpret procedures, plus a control processor, which performs scheduling and controls the multiplexing of the instruction processors. The memory module implements the virtual memory. The memory module interface consists primarily of invocations of the vector operations.

The memory module consists of the vector module and the page module. The vector module implements automatic storage reclamation. It consists of a number of vector processors, which perform vector operations requested by the processing module, plus associated reference count processors, which perform reference count operations resulting from vector operations, plus a reclamation processor, which processes the GQs and performs garbage collection.

The page module implements the basic virtual (multi-level) memory. Its interface consists of operations of the page data type. The page module consists of the page handler, the storage processor, the secondary storage module, plus a conventional primary storage module. The page handler maps page references (secondary storage addresses) to primary storage addresses and initiates swapins of needed pages not in primary storage. It has exclusive access to the mapping data base, which is the set associative memory. The page handler receives requests from the vector module via a multiplexor, which arbitrates incoming requests and presents them one at a time to the page handler, which processes requests sequentially.

The storage processor performs primary storage allocation and deallocation at the request of the page handler and initiates page replacement. It has exclusive access to the primary storage allocation data base. The secondary storage module performs secondary storage allocation and deallocation and performs transfers between primary storage and secondary storage. It has exclusive access to the secondary storage allocation data base.

Splitting the system into modules in this manner leads to an organization that is superior to conventional systems constructed out of a single general purpose processor (or a small number of general purpose processors) and a complex operating system kernel. The system can be understood as a collection of modules communicating via messages. The interfaces between modules are simple, often corresponding to operations of abstract data types. There are no timing constraints on the speed at which messages are transmitted or acted upon. Each module can be understood individually in terms of its interface to the rest of the system. Many modules encapsulate important data bases (e.g., the page map) and can easily control and synchronize accesses to those data bases.

While this or any multiprocessor organization can be simulated using multiple processes on a single processor, using separate hardware modules has a number of advantages. The most obvious advantage is better performance: multiple processors allow true concurrency, which can increase the throughput of the system. Furthermore, any simulation of multiple processors will involve overhead. In our system, the rate of interaction between some modules is quite high; the overhead of simulating this behavior could be substantial. Another advantage is better isolation between modules. The only connections between modules are the message channels; modules can interact only in well defined ways via message passing. In a uniprocessor simulation, interprocess isolation must be *proven*. Proper isolation is difficult to achieve without hardware support. Even with hardware support, one must be careful to avoid interactions via the process scheduler. To maximize performance, uniprocessor systems generally use multiple priority levels, preemption, and interrupts. In such systems, additional potential exists for problems of unfair scheduling and deadlock not present in our multiprocessor organization.

9.2 Further Work

The biggest question left unanswered by this thesis is, will it work? If the proposed system were constructed, would its performance be adequate? There are two methods that could be used to answer this question. One way, of course, is to actually construct a system. The other way is to use modeling and simulation to estimate the performance of the system. This latter method is preferable if it can produce meaningful results using less time and resources than would be required to actually construct a machine. The difficulty is that the proposed system is sufficiently different from current systems (in particular, in providing a single large virtual memory consisting of sharable small objects) that predictions based on data derived from conventional systems are likely to be unreliable. It is difficult to predict how users will use an unconventional system short of giving them one and observing the results. Obtaining relevant data without building a real system would probably require a fairly elaborate simulation of the proposed architecture.

Both of these methods represent a substantial undertaking and are clearly outside the scope of a single thesis. What we have done instead in this thesis is to give plausibility arguments to show at least that we are in the right ballpark. Wherever possible, we have based our performance estimates on limited data obtained from an existing single process, small address space implementation of CLU and on data from related systems. We do not expect the reader to believe that the system will perform adequately; however, we do hope to convince the reader that further investigation of this kind of machine architecture would be worthwhile.

There are a number of areas where the proposed design is incomplete. For example, the machine language has not been fully specified, particularly in the area of control structures. No provision has been made for I/O. A number of issues relating to the interface between the machine language and user languages (e.g., CLU) have not been explored, such as support for debugging and mechanisms for linking procedures together. If a real system were to be constructed, one would also want to consider providing more hardware support for specific language features, e.g., extendible arrays.

One major area that requires further investigation is the subject of resource allocation and control. There are a number of ways in which the gross performance of the system is dependent upon the behavior of individual processes. Of particular importance in this system are the rate of generation of cyclic garbage, which affects the rate at which garbage collection is required, and the variability of the distribution of object sizes, which affects the amount of secondary storage fragmentation and can force the system to be stopped to perform compaction. Of course, as in any system, the total storage usage is also important. Thus, an individual process, by its actions, could force the system into a state where new objects could not be created, thus interfering with the ability of other processes to execute successfully.

In many applications, this situation would be unacceptable. The solution must be some mechanism to limit the ability of an individual process (or user) to acquire storage resources. The mechanism must take into account secondary storage made unavailable by cyclic garbage or external fragmentation. There are two reasons why resource allocation is a more difficult problem in an object-oriented system, as opposed to a more conventional system: One reason is that objects can be shared; it is difficult to assign responsibility for a shared object in a fair manner. The other reason is that objects are large in number and generally small in size. Thus, traditional methods may not be acceptable because of excessive overhead per object.

Another area requiring further investigation is the issue of off-line storage, both for the purposes of reliability and archiving. When an object is copied onto off-line storage, how much of the graph of objects accessible from that object should be written with it? (Suppose an object contains a reference to the root of the file system - why not! Should the entire file system be copied onto off-line storage with it?) When the object and any associated objects are read back into the system from off-line storage, how are the objects re-integrated with the existing collection of objects? How is sharing preserved (or is it)?

These problems are not unique to the proposed architecture, but are problems that must be faced in any object-oriented system. Moreover, these problems also appear in conventional systems. It's just that the problems are more obvious and less easily solved in object oriented systems where small objects are directly supported and where sharing and inter-object dependencies are made explicit in the form of object references.

9.3 A Perspective

It is important to recognize the relationship between effective system design and the characteristics of available technology. Our design is predicated on the assumption that the desired virtual memory size is consistent with affordable amounts of storage devices that provide relatively fast access to small pieces of data. This situation is not true today, but will likely become true with the development of faster access secondary storage devices. How long this situation will then persist depends upon whether the need for larger and larger virtual address spaces grows faster or slower than the ability of technology to provide larger fast secondary storage devices. We will not attempt a prediction here.

It is relevant to consider the relationship between this work and the growing popularity of "distributed computing." One likely form of a distributed computing system would provide separate local address spaces at each node, in addition to a distinct mechanism to support global addressing. We can easily imagine our proposed system as a node providing a moderate sized (10-100M word) local address space to a small group of cooperating users. We can also conceive of our system being used as a rather powerful personal computer. (Even a single user can utilize multiple concurrent processes.) While a multiprocessor implementation as we have proposed is currently too expensive for either of these uses, it seems clear that such will not always be the case.

Appendix I - A Possible Machine Language

In this section we describe a very simple machine language that could be used to specify user defined procedures. This proposal omits a number of features that would be needed to support CLU, e.g., exception handling and iterators. It also ignores many possibilities for optimization.

1. Procedures

A procedure consists of two parts. One part, the *code section*, consists of an addressable collection of *instructions*. A set of possible instructions is described below. The second part, the *linkage section*, consists of an addressable collection of arbitrary references. Included in the linkage section would be (references to) all literals needed by the procedure, as well as other procedures invoked by the procedure. We assume that the code section and the linkage section are both addressed by small, nonnegative integers.

2. Procedure Activations

A procedure activation is a collection of information that represents one particular invocation of a user defined procedure. The information contained in a procedure activation includes the following:

- The procedure being interpreted.

- An instruction counter.

  The instruction counter is an integer that identifies one particular instruction in the code section of the procedure being interpreted. That instruction is the one currently being interpreted or about to be interpreted.

- A stack frame.

  The stack frame is an addressable collection of arbitrary references. It contains the actual arguments of the procedure invocation, local variables of the procedure, plus temporaries. The stack frame is addressed by consecutive integers, starting with zero. Elements can be added or deleted from the high end (the *top*) of the stack frame.

3. Instruction Set

The following instructions are allowed as part of the code section of a user defined procedure. Instruction operands are written in italics. All operands are small nonnegative integers; the interpretation of operands is dependent on the particular instruction. We assume that all accesses to code sections, linkage sections, and stack frames are checked for attempts to access outside the current bounds of the collection.

The interpretation of instructions proceeds "sequentially" unless a specific next instruction is specified by a branch instruction.

- **apply** *nargs*

  The **apply** instruction causes a procedure to be invoked. The top element is popped off the stack frame; this object must be a procedure object. *Nargs* elements are removed from the top of the stack frame; these objects will be the actual arguments of the invocation (the first argument is lowest on the stack). The specified procedure is then invoked with the specified argument objects (see below); when the procedure terminates, the result objects are left on the top of the stack frame.

  When a user defined procedure is invoked, a new procedure activation is created, including a new stack frame. The new stack frame is initialized to contain the actual arguments. Interpretation of the code section begins with the initial instruction.

When a primitive procedure is invoked, no procedure activation is created. Instead, the machine directly computes the result objects and pushes them on the caller's stack frame. In the case of primitive procedures, the machine also checks each actual argument object to make sure it is of the expected type (one of vector, bstring, procedure, or process).

- **return** *nobjs*

  The **return** instruction terminates the execution of a user defined procedure. *Nobjs* specifies the number of return objects, which are removed from the top of the stack frame. These objects are pushed onto the top of the calling activation's stack frame and interpretation of the calling activation is resumed.

- **pushstack** *addr*

  The **pushstack** instruction copies a reference from the stack frame element specified by *addr* and pushes it onto the top of the stack frame.

- **pushlink** *addr*

  The **pushlink** instruction copies a reference from the linkage section element specified by *addr* and pushes it onto the top of the stack frame.

- **pop** *addr*

  The **pop** instruction pops a reference from the top of the stack frame and stores it in the stack frame element specified by *addr*.

- **branch** *addr*

  The **branch** instruction causes interpretation to continue with the code section element specified by *addr*.

- **fbranch** *addr*

  The **fbranch** instruction pops a reference from the top of the stack frame. If that reference is not true, then control branches to the code section element specified by *addr*.

4. Notes

A number of representation details have not been specified, for example: how procedures are represented, how instructions are encoded, how procedure activations are represented, and how the procedure activations of a single process (including all stack frames) can be implemented using a single stack object. These details (if done correctly) do not affect the semantics of the machine language; they are invisible to the machine language program.

5. Example

Figure 24 shows a factorial procedure written in CLU, along with the corresponding machine language program. Symbolic names have been used for addresses into the stack frame, control section, and linkage section, in the traditional assembly language manner.

Figure 24. An example of the machine language.

Factorial in CLU:

```
fact = proc (n: int) returns (int)
        f: int := 1
        while n > 1 do
                f := f * n
                n := n - 1
                end
        return (f)
        end fact
```

Stack offsets:

```
n = 0    % the argument to fact
f = 1    % a local variable
```

Linkage section:

```
one:     "the integer 1"
gt:      "the procedure int$gt (greater than)"
mul:     "the procedure int$mul (multiply)"
sub:     "the procedure int$sub (subtract)"
```

Code section:

```
     pushlink one
L1:  pushstack n; pushlink one; pushlink gt; apply 2; fbranch L2
     pushstack f; pushstack n; pushlink mul; apply 2; pop f
     pushstack n; pushlink one; pushlink sub; apply 2; pop n
     branch L1
L2:  return 1
```

Appendix II - Page Module Description

This appendix contains brief descriptions of the operations of three submodules of the page module: the page handler, the storage processor, and the secondary storage module. For each module, we list the possible requests that may be received by the module and a description of the action taken upon receiving that request. A message is written as command(arg1,arg2,...), where *command* is the request name and *arg1*, *arg2*, etc., are message arguments.

1. Legend

rid = request port identification (from Vector Module)
SPID = special rid indicating Storage Processor
sn = size number (encodes the page size)
sp = secondary storage page address
pp = primary storage page address
NILP = special NIL page id, in page map entry, indicates that the page is being swapped in
mod = a boolean indicating that the primary storage copy of a page has been modified
setn = a set number in the set associative memory
oper = [fetch,store,incr_rc,decr_rc,mark,unmark]

2. Page Handler (PH)

- **alloc(sn,rid)**

```
send alloc(sn,rid) to SP
```

- **new(sp,pp,rid)**

```
enter [sp,pp] in page map
if entry pushed out, send swap_out(osp,opp,omod) to SP
send result(sp) to RID
```

- **equal(sp1,sp2,rid)**

```
compare the addresses sp1 and sp2
send result(eq) to RID
```

- **oper(sp,rid,...)**

```
lookup page map entry
if no entry then
  queue request
  enter [sp,NILP] in page map
  if entry pushed out, send swap_out(osp,opp,omod) to SP
  send swap_in(sp) to SP
elseif e.pp=NILP %in_transit% then
  queue request
else
  perform request
  send result(...) to RID
end
```

- **swap_in_done(sp,pp)**

```
update page map entry with pp %not in_transit%
perform queued requests
```

- **touch(sp,rid)**

```
lookup page map entry
if no entry then
  enter [sp,NILP] in page map
  if entry pushed out, send swap_out(osp,opp,omod) to SP
  send swap_in(sp) to SP
  send result(false) to RID
elseif e.pp=NILP %in_transit% then
  send result(false) to RID
else
  send result(true) to RID
end
```

- **replace(setn)**

```
select page to replace from the specified set of the page map
remove entry from page map
send swap_out(sp,pp,mod) to SP
```

- **dealloc(sp,rid)**

```
lookup page map entry
if no entry then
  send dealloc(sp,NILP) to SP
else
  remove entry from page map
  send dealloc(sp,pp) to SP
end
```

- **size(sp,rid)**

  send size(sp,rid) to SP

- **sweep_reset(rid)**

  send sweep_reset(rid) to SP

- **sweep_next(rid)**

  send sweep_next(rid) to SP

3. Storage Processor (SP)

- **alloc(sn,rid)**

  send alloc(sn,rid) to SSM

- **new(sn,sp,rid)**

  allocate primary storage page of given size
  initialize primary storage page
  send new(sp,pp,rid) to PH

- **swap_in(sp)**

  send size(sp,SPID) to SSM

- **size(sp,rid)**

```
send size(sp,rid) to SSM
```

- **size_is(sp,sn,rid)**

```
if rid=SPID then % continuing swap_in
  allocate primary storage page of given size
  send swap_in(sp,pp) to SSM
else
  send result(sn) to RID
end
```

- **swap_in_done(sp,pp)**

```
send swap_in_done(sp,pp) to PH
```

- **swap_out(sp,pp,mod)**

```
if ~mod then deallocate primary storage page
else send swap_out(sp,pp) to SSM
```

- **swap_out_done(sp,pp)**

```
deallocate primary storage page
```

- **dealloc(sp,pp)**

```
send dealloc(sp) to SSM
if pp~=NILP then deallocate primary storage page
```

- **sweep_reset(rid)**

    flush page buffer (so that all free pages have identifiable free list
      pointers in their header words)
    reset internal sweep counter
    send result() to RID

- **sweep_next(rid)**

    get sp of next page to be yielded (update internal counter),
      sp=NILP if no more
    send result(sp) to RID

4. Secondary Storage Module (SSM)

- **alloc(sn,rid)**

    allocate secondary storage page of given size
    send new(sn,sp,rid) to SP

- **size(sp,rid)**

    determine size of page
    send size_is(sp,sn,rid) to SP

- **swap_in(sp,pp)**

    transfer page from secondary storage to primary storage
    send swap_in_done(sp,pp) to SP

- **swap_out(sp,pp)**

  transfer page from primary storage to secondary storage
  send swap_out_done(sp,pp) to SP

- **dealloc(sp)**

  deallocate secondary storage page

References

1. Ackerman, W. B. A structure memory for data flow computers. Rep. TR-186, M.I.T. Laboratory for Computer Science, 1977.

2. Baker, Jr., H. G. Actor systems for real-time computation. Rep. TR-197, M.I.T. Laboratory for Computer Science, 1978.

3. Batson, A. P. and Brundage, R. E. Segment sizes and lifetimes in Algol 60 programs. *Comm. ACM 20*, 1 (Jan. 1977), 36-44.

4. Bawden, A., et. al. LISP machine progress report. Memo 444, M.I.T. Artificial Intelligence Laboratory, 1977.

5. Bensoussan, A., et. al. The Multics virtual memory: concepts and design. *Comm. ACM 15*, 5 (May 1972), 308-318.

6. Birtwistle, G. M., et. al. *Simula begin.* Auerbach, Philadelphia, 1973.

7. Bishop, P. B. Computer systems with a very large address space and garbage collection. Rep. TR-178, M.I.T. Laboratory for Computer Science, 1977.

8. Clark, D. W. List structure: measurements, algorithms, and encodings. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University, 1976.

9. Conti, C. J. Concepts for buffer storage. *IEEE Computer Group News 2*, 8 (Mar. 1969), 9-13.

10. Denning, P. J. The working set model for program behavior. *Comm. ACM 11*, 5 (May 1968), 323-333.

11. Dennis, J. B. First version of a data flow procedure language. *Lecture Notes in Computer Science 19*, Springer-Verlag, N. Y., 1974, 362-376.

12. Deutsch, L. P. and Bobrow, D. G. An efficient, incremental, automatic garbage collector. *Comm. ACM 19*, 9 (Sept. 1976), 522-526.

13. Dijksta, E. W., et. al. On-the-fly garbage collection: an exercise in cooperation. *Comm. ACM 21*, 11 (Nov. 1978), 966-975.

14. Fabry, R. S. Capability-based addressing. *Comm. ACM 17*, 7 (July. 1974), 403-412.

15. Gries, D. An exercise in proving parallel programs correct. *Comm. ACM 20*, 12 (Dec. 1977), 921-930.

16. Hatfield, D. J. Some experiments on the relationship between page size and program access patterns. *IBM JRD 16*, 1 (Jan. 1972), 58-66.

17. Hewitt, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence 8*, 3 (June 1977), 323-364.

18. Kaehler, T. Private communication, 1978.

19. Knuth, D. E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Ma., 1968.

20. Lampson, B. W. Protection. Proc. Fifth Princeton Symposium on Information Sciences and Systems, Princeton University, March 1971, 437-443. Reprinted in *Operating Systems Review 8*, 1 (Jan. 1974), 18-24.

21. Liskov, B., et. al. Abstraction mechanisms in CLU. *Comm. ACM 20*, 8 (Aug. 1977), 564-576.

22. Liskov, B., et. al. CLU reference manual. Computation Structures Group Memo 161. M.I.T. Laboratory for Computer Science, 1978.

23. Liskov, B. and Snyder, A. Structured exception handling. Computation Structures Group Memo 155-1. M.I.T. Laboratory for Computer Science, 1978.

24. Madnick, S. E. Storage Hierarchy Systems. Rep. TR-107, M.I.T Project MAC, 1973.

25. Martin, R. R. and Frankel, H. D. Electronic disks in the 1980's. *Computer 8*, 2 (Feb. 1975), 24-30.

26. Mattson, R. L., et. al. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78-117.

27. Reed, D. P. Processor multiplexing in a layered operating system. Rep. TR-164, M.I.T. Laboratory for Computer Science, 1976.

28. Saltzer, J. H. Traffic control in a multiplexed computer system. Rep. TR-30, M.I.T. Project MAC, 1966.

29. Schorr, H. and Waite, W. M. An efficient machine-independent procedure for garbage collection in various list structures. *Comm. ACM 10*, 8 (Aug. 1967), 501-506.

30. Schroeder, M. D. Performance of the GE-645 associative memory when Multics is in operation. Proc. ACM Workshop on System Performance Evaluation (April 1971), 227-245.

31. Smith, A. J. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering SE-4*, 2 (Mar. 1978), 121-130.

32. Steele, Jr., G. L. Multiprocessing compactifying garbage collection. *Comm. ACM 18*, 9 (Sept. 1975), 495-508.

33. Theis, D. J. An overview of memory technologies. *Datamation 24*, 1 (Jan. 1978), 113-131.

34. Van Wigngaarden, A., et. al. Revised report on the algorithmic language Algol 68. *Sigplan Notices 12*, 5 (May 1977), 1-70.

35. Wadler, P. L. Analysis of an algorithm for real time garbage collection. *Comm. ACM 19*, 9 (Sept. 1976), 491-500.

36. Weisman, Clark. *Lisp 1.5 Primer*. Dickenson Press, 1967.

37. Weng, K-S. An abstract implementation of a generalized data flow processor. Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., forthcoming.