

**A DATABASE AS THE BASIS OF AN OPERATING
SYSTEM**

Jeffrey Schaffer

**June 1986
CSD-860037**

UNIVERSITY OF CALIFORNIA

Los Angeles

A Database as the Basis of an Operating System

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in Computer Science

by

Jeffrey Paul Schaffer

1986

© Copyright by
Jeffrey Paul Schaffer
1986

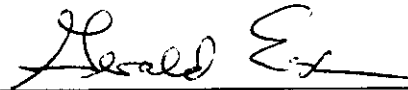
The thesis of Jeffrey Paul Schaffer is approved.



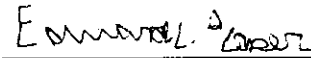
Daniel M. Berry



Mario Gerla



Gerald Estrin, Committee Co-Chair



Edward L. Glaser, Committee Co-Chair

University of California, Los Angeles

1986

TABLE OF CONTENTS

1. Introduction.....	1
2. Relational Databases.....	4
2.1. The Relational Model.....	4
2.1.1. Relational Structures.....	5
2.1.2. Relational Operations.....	8
2.2. Relational Programming.....	11
2.2.1. Description of the Language.....	12
2.2.2. Example – Implementing Relational Model Operations.....	20
3. Relational Implementation of Data Structures.....	23
3.1. Stacks.....	24
3.2. Queues.....	28
3.3. Deques.....	32
3.4. Lists.....	34
3.5. Arrays.....	37
3.6. Trees.....	39
4. Operating System Implementation.....	46
4.1. Physical Devices.....	46
4.2. Processes.....	48
4.2.1. Process Creation.....	49
4.2.2. Process Interaction.....	51
4.2.2.1. Signals.....	52
4.2.2.2. Semaphores.....	53
4.2.2.3. Mailboxes.....	55
4.2.3. Process Scheduling.....	56
4.2.3.1. Round-Robin.....	57
4.2.3.2. Multi-Level Feedback.....	58
4.3. Virtual Memory.....	60
4.3.1. Memory Allocation.....	61
4.3.2. Paging.....	63
4.3.3. Page Replacement.....	64
4.4. Filesystems.....	66
5. Conclusions.....	69
5.1. The Real World.....	70
5.2. Suggestions for Further Work.....	71
Bibliography.....	73

LIST OF FIGURES

2.1. A Typical Relation and its Components.....	8
3.1. A Stack.....	24
3.2. Relational Implementation of a Sequence.....	25
3.3. A Queue.....	29
3.4. Improper Data Insertion in a Queue.....	32
3.5. Indexed Representation of a Sequence.....	35
3.6. Relational Representation of a Two-dimensional Array.....	39
3.7. Relational Implementation of a Tree.....	42
4.1. Physical Device Relations.....	47
4.2. Process Hierarchy Relation.....	49
4.3. Signalling Relation.....	53
4.4. Semaphore Relations.....	54
4.5. Mailbox Relation.....	56
4.6. Process Scheduling Relations.....	58
4.7. Virtual Memory Relations.....	61
4.8. LRU Relation.....	65
4.9. Filesystem Relations.....	67

ACKNOWLEDGEMENTS

I want to thank my family, friends, and committee for your help during the writing of this thesis; your encouragement, direction, and occasional boots in the behind were all necessary in order for me to complete the paper. In addition, two people deserve special thanks. First, to Ted Glaser, my advisor and my friend, for teaching me to view the world differently and for first suggesting the subject. Second, to my wife JoEllen, for your editing, proofreading, assistance, and general support – though your patience with me often wore thin, your were always there when I needed you. I could not have completed this without you.

ABSTRACT OF THE THESIS

A Database as the Basis of an Operating System

by

Jeffrey Paul Schaffer

Master of Science in Computer Science

University of California, Los Angeles, 1986

Professor Gerald Estrin, Co-Chair

Professor Edward L. Glaser, Co-Chair

This paper shows that an operating system can be constructed such that its kernel consists of a database system, unifying all of the data structures within the computer. This concept is demonstrated by implementing key operating systems concepts on top of a database; from these concepts a complete operating system can be built. The results show that this type of underlying structure provides for a simplified operating system, easily built and modified with a minimum of programming effort. The results also show that a production version of such a system is impractical given current computer architectures; full implementation would require a customized architecture. Given this, an alternative use of such a system is provided.

Chapter One

Introduction

Database systems are beginning to be recognized as powerful programming tools; by providing a single, consistent view of data and a pre-defined access strategy, a database system eases the program development task. As a result, many application programs are being developed which incorporate a database system as the underlying storage mechanism, significantly reducing the time and effort involved in developing these programs [Martin82]. Such methods have never been applied, however, to operating systems development. It is the thesis of this paper that an operating system can be constructed such that its kernel consists of a database system, unifying all of the data structures within the computer. Further, that this type of underlying structure provides for a simplified operating system, easily built and modified with a minimum of programmer effort.

Can an operating system benefit from a database underpinning? The answer is "yes", for like any application program an operating system is an information processor. In this case, the database consists of information on the state of the system and on the programs and data contained within the system; this information is then applied as necessary to manage the system. In a conventional setup, all of this data is stored in

separate sets of hard-coded structures. Two problems exist with this method of data organization: (1) each structure requires its own access method, and (2) structures are not easily extendable when additional information is required. In a system with an underlying database, however, only one type of data structure exists with one uniform access strategy; this permits the programmer to concentrate on the algorithm to be implemented rather than the data manipulation mechanisms required to program it. If another type of structure is desired, it is *logically*, not physically, defined on top of the existing database structure. This makes the structure easily extended as well.

This paper will show that a database can serve as the basis of an operating system by implementing key operating systems concepts on top of a database; from these concepts a complete operating system can be built. Since this paper uses the relational data model for its underlying storage facility, the operating system being implemented will be referred to as a *relational operating system*; this is used in preference to the term *database operating system* in order to avoid confusion with any existing literature [Gray78].

Before attempting to implement the operating system, it should be noted that one strong argument can be raised against any database oriented operating system — performance. If executed under existing computing architectures, a relational operating system will simply be slow. Since the purpose of this paper, however, is only to prove that a relational operating system is possible, two assumptions will be made: (1) that the implementation machine is of a custom architecture providing the appropriate support, and (2) that this machine is sufficiently fast. The impact of these assumptions will be discussed in the conclusions section of this paper.

The remainder of this thesis is divided into four chapters. Chapter 2 provides background material on relational databases and the relational programming language that will be used to implement the operating system. Chapter 3 demonstrates how relational programming can be used to implement other forms of data structures; e.g. stacks, queues, etc. This chapter also serves as an example in relational program development, documenting the evolution of each data structure from concept to final implementation. Chapter 4 is the heart of the paper, describing numerous operating system concepts and showing how these are implemented using an underlying database. Finally, Chapter 5 presents concluding comments and suggests areas for further research.

Chapter Two

Relational Databases

This chapter provides the reader with a brief background on the data model and programming language chosen for implementation. Though many different data models exist, the *relational model* is chosen for its simplicity; the structures provided by this model form a natural representation for the data and are easily manipulated. The choice of a programming language, on the other hand, is more limited; few relational languages exist which do not require support from a conventional programming language. Thus, the programming language chosen is one which does not require any outside support.

This chapter is organized into two sections. The first section discusses the relational model, its data structures and operations. The second section discusses the concept of relational programming and describes the specific relational programming language chosen for implementation.

2.1. The Relational Model

First proposed by Codd in [Codd70], the relational model is based on the theory that data can be expressed in the form of a simple data structure which exhibits the properties of a mathematical relation; i.e. a structure which applies elementary relational

theory for data manipulation. The primary advantage of this approach is simplicity; only one type of data structure exists at the external (user) level in a form which constitutes a natural representation for data. Unlike the hierarchical or network model, the relational data structure is independent of the underlying physical storage system; this permits changes in the storage structure and/or the data access strategies without modifying the user's programs or query techniques. The absence of a rigidly defined storage structure also permits separation of the model's semantics from its structure; since no underlying structure is implied, the eventual implementation structure chosen has no effect on the data manipulation operations as is often true in other data models.

This section presents a brief description of the data structures and operations which define the relational model. Additional information on the model, including complete descriptions of its theory and restrictions, is presented in [Chamberlin76], [Codd70], [Date81], [Maier83], and [Ullman82].

2.1.1. Relational Structures

The fundamental structure in any database system, relational or otherwise, is the database itself. A *database* is a stored, time-varying collection of data available for computer processing. While only a single database is necessary, relational database systems often support multiple databases for reasons of security, physical storage limitations, and data transportability. In a multiple database arrangement, each database is considered independent from the others; this limits data manipulation operations to operating only on data contained within a single database at any one time.¹ Since the

¹Generally, however, a small set of operations is provided for copying data between databases for further relational processing.

concepts of a relational database are the same regardless of the number of databases supported, this paper assumes the existence of only a single database.

Within a database, data elements are divided into sets by their datatype; these sets are known as *domains*. A domain can be thought of as a pool of values from which data is drawn; typical domains include integers, names, addresses, zip codes, etc. Each domain is restricted to include only those data values considered valid for that domain, with validity requirements established at the time the domain is defined. A domain has the property that it is separate and distinct from all other domains in the same database; e.g. data drawn from the domain *zip code* is considered distinct from data drawn from the domain *integer*, regardless of the fact that *zip code* may be defined as a subset of *integer*.

When a domain is defined, the list of data elements valid for that domain may either be explicitly or implicitly specified; for example, a domain of part names would be explicitly defined, while a domain of real numbers (containing an infinite number of members) would be implicitly defined. While in theory a database should contain all of the data values permissible in each domain, it is not practical to store those values not currently in use due to physical storage limitations. In order to deal with this, databases are restricted to contain only those values contained in each *active domain*; an active domain is that portion of a domain that is currently in use.

Active domains, or subsets of active domains, can be combined into a set of n-tuples which represent all of the valid combinations between the data elements; this set is referred to as a *relation*.² A relation describes the relationship which exists between all of the domains after combination; specifically, individual data elements within each tuple exhibit a direct relationship to all other data elements in that same tuple. As an example, if

²An n-tuple is generally referred to simply as a *tuple*; for convenience, this term will be used for the remainder of this paper.

a relation is the combination of the domains *name*, *address*, and *zip code*, then each tuple in that relation correlates a name with a specific address and zip code. Note that this relationship is bi-directional; each tuple in the relation also associates the address or zip code in that tuple only with a specific name.

Physically a relation resembles, and is often represented by, a table consisting of an arbitrary number of rows and columns; figure 2.1 shows a typical relation in table form.³ The number of rows in the relation refers to the *cardinality* of the relation, while the number of columns refers to the *degree* of the relation. Each column in a relation is an *attribute* of that relation; an attribute represents that set of data elements drawn from the domain which defines the attribute. Associated with each attribute is an *attribute name*; this name must be unique within the relation itself but not necessarily within the database. Related data elements from all of the attributes involved in a relation form each tuple of that relation; this is the equivalent of a row in the table model. Attribute order within each tuple is fixed; the rearrangement of attributes in only a subset of a relation's tuples produces a relation of radically different meaning. Tuple order in a relation, however, is arbitrary since every tuple represents a unique combination of data elements; positional reassignment of that tuple has no effect on its meaning.

³Formally a table is a restricted view of a relation; for this discussion, however, it is an appropriate and adequate model.

Relation *personnel*

<u>name</u>	<u>address</u>	<u>home phone</u>	<u>office phone</u>
John Smith	921 Applecore	555-9452	825-9452
Mary Doe	225 Sierra Park	555-1025	825-9999
...			

Attribute names: *name*, *address*, *home phone*, *office phone*

Attribute *name* drawn from domain *names*, a subset of *alphanumeric*.

Attribute *address* drawn from domain *alphanumeric*.

Attribute *home phone* drawn from domain *phone numbers*.

Attribute *office phone* drawn from domain *phone numbers*.

domain <i>name</i>	{John Smith, Mary Doe, Jeff Schaffer, ...}
domain <i>alphanumeric</i>	{a, b, ..., ab, abc, ..., 921 Applecore, ...}
domain <i>phone number</i>	{555-1025, 555-9452, 825-1234, ...}

Figure 2.1. A Typical Relation and its Components

2.1.2. Relational Operations

Since a relation is in fact a set of tuples, then all of the standard set manipulation operations can be applied; specifically, the operations *union*, *intersection*, *difference*, and *negation* are all valid on relations. Three of these operations, union, intersection, and difference, are dyadic operations and require that both input relations be of identical structure; i.e. the attributes of both relations must be in identical order with each attribute pair (one from each relation) derived from the same domain. These operations result in a relation of the same structure as the input relations.

The final set operation, negation, is a monadic rather than dyadic operation; this operation, however, also produces a result relation which is identically structured to the input relation. Normally negation is defined as the difference between the relation formed by the *Cartesian product* of the domains of the input relation and the input relation itself.⁴ The problem with this definition occurs when one of the domains of the input relation contains an infinite number of values; negating this relation results in an infinite relation, which is not strictly defined. To correct this problem, the Cartesian product is taken over the active domain of the relation only, assuring a result relation with a finite number of tuples [Maier83].

In addition to the basic set manipulation operations, four special operations exist for relations; these operations, however, need not produce a result which is identical in structure to the input relation(s). These special operations are:

1. *Select* – this operation produces a result relation which consists of a subset of tuples from the input relation; membership in the resulting subset is based on whether the individual tuple meets a user specified criteria. The result relation produced by the ‘Select’ operation consists of the same number of attributes as the input relation, but only a subset of the tuples.
2. *Project* – this operation produces a result relation which consists of a subset of attributes from the input relation. The result relation produced by the ‘Project’ operation consists of the same number of tuples as the input relation, but only a subset of the attributes.
3. *Join* – this operation produces a result relation which is the concatenation of a subset of tuples from the first input relation with a subset of tuples from the second input relation. Tuples are selected from each relation on the basis that, for

⁴The Cartesian product of n domains defines the maximum relation that can hold between those domains.

user specified attributes in each input relation defined over the same domain, data elements common to both relations exist in these attributes.⁵ Although this operation is generally defined only on two attributes (one per relation), the definition of a join does not restrict the number of attribute pairs on which the join may occur provided that each attribute pair is defined over the same domain. The number of attributes in the result relation produced by the 'Join' operation is the sum of the number of attributes of each input relation; the number of tuples in the result relation is variable and depends on the number of shared data elements between the relations.

4. *Update* – this operation produces a result relation identical to the input relation except at selected row-column intersections; at these intersections the individual data elements have been modified. The result relation produced by the 'Update' operation is identical in structure to the input relation.

Operations which insert and delete tuples in a relation are unnecessary since this can be performed by the set operators 'union' and 'select', respectively. While other data manipulation operations exist, those described above are considered primary and the only ones of concern in this paper. Missing still, however, are the structure definition operations; these operations create and delete relations and domains. Creating a relation involves (1) creating an ordered list of attributes, (2) specifying the domains associated with those attributes, and (3) naming the new relation. Creating a domain involves (1) naming the domain, and (2) defining the valid values for the domain (either explicitly or implicitly). Deleting a relation destroys both the defined structure and the data contained within it. Deleting a domain, however, destroys the data within it only when the domain

⁵This type of join is known as a *equijoin* since the joining condition is based on data element equality; similar definitions exist for other types of join, e.g. not equal, less than, greater than, etc.

is no longer associated with any attribute in the database; until this occurs deletion requests are ignored.

2.2. Relational Programming

This section discusses *relational programming*, a type of programming in which relations constitute the primary data structure used for computation and manipulation. This concept is similar to that of functional programming [Backus78], in which functions form the principle object manipulated. Like functional programming, relational programming has many advantages over traditional languages; it also, however, offers several advantages over functional programming itself. Some of these advantages, as originally noted in [MacLennan83], are:

1. There is no distinction between functions and data: in relational programming, a function is represented in the same form as other data – as a relation. This provides for a concise and powerful language since only a single set of data operators is required for both the manipulation of functions and data.
2. Multi-valued functions are valid: since a function is a relation, it is perfectly valid to permit any function to be multi-valued in either input or output; i.e. any function can be one-to-one, one-to-many, many-to-one, or many-to-many.
3. Relations obey simple laws: the rules which govern relations are less restrictive than those governing functions. For example, the property

$$(f.g)^{-1} = g^{-1}.f^{-1}$$

is true only for one-to-one functions, but is true for all relations.

4. Relations can represent complex data structures: relations can model complex non-linear data structures, such as graphs, as easily as they can simpler linear structures.

Finally, as in functional programming, relational programming languages have also been shown to be complete programming languages [Kowalski78, MacLennan83]. This, along with the advantages cited, led to the selection of a relational programming language as the implementation mechanism for the relational operating system.

The remainder of this chapter discusses the relational programming language proposed in [MacLennan81, MacLennan82, MacLennan83]. Note that the language presented in this paper represents a combination of the different notations presented in each of the papers listed above, with modifications introduced as necessary for clarity; regardless of the notation, however, the concepts described in this paper are identical to those presented in the original papers.

2.2.1. Description of the Language

Relational programming deals with three types of data objects: *individuals*, *classes*, and *relations*. The first of these objects, the individual, consists of any singular data value; e.g. a specific number or a particular name. In relational programming, an individual represents the smallest manipulable data object. The second of the data objects, the class, is a collection of individuals, i.e. a set. Often, though not always, individuals are grouped into classes by a common property; e.g. the class *integer* consists of a group of individual numbers all of which share of property of being integers. Thus, classes are analogous, but not identical, to the concept of a domain in the relational model; the definition of a domain mandates that a common property relate all elements in the domain, while a class may consist of unrelated objects. Since the size of a class is variable, the actual number of objects contained in the class at any one time is determined

by the relational operator 'size'; this operator, denoted as **size** C , returns an integer indicating the current size of the class.⁶

The final data object, the relation, describes the correlation between the members of two, not necessarily distinct, classes. In a relation, each member of the first class is associated with one or more members of the second class; thus a relation is actually a set of pairs where each pair describes the relationship between the classes. In relational programming, a relationship is described by the notation xRy , indicating that element x in class C_1 bears the relationship R to element y in class C_2 . As an example, the notation $5 < 10$, where both elements are members of the class *integer*, signifies that element 5 exhibits the less than relationship to element 10. This definition of a relation differs from that of the relational model by its restriction to a two element, or binary, relation; this does not prove to be a functional restriction, however, since any general relation can be decomposed into a comparable set of binary relations [Kowalski78].

A relation may also be viewed as a two column table, with each row representing a pair between the classes. The elements in the first, or leftmost, column bear the stated relation to the corresponding elements in the second, or rightmost, column. As an example, if there exists a class consisting of the elements {1, 3, 5, 7}, then forming the relationship '<' on this class yields the table:

<	
1	3
1	5
1	7
3	5
3	7
5	7

⁶The symbols R and C will be used throughout the remainder of this paper to denote a relation and a class, respectively.

Each column is a subset of one of the original classes which formed the relation. The class of elements contained in the left column of the relation is referred to as the *domain* of the relation, while the class formed by the right column is referred to as the *range* of the relation. The relational operators **domain** and **range** are used to separate the respective classes from the relation; each operator extracts the proper column and deletes duplicate values.

The inverse of a relation is described by the notation $xR^{-1}y$, indicating that each member of the range of the relation is associated with one or more elements of the domain. In the table representation of a relation this is equivalent to reversing the columns of the table. Using the ' $<$ ' relation above as an example, it can be seen that the inverse of the relation yields the ' $>$ ' relation.

Relations can be combined by the standard set operators intersection, union, and difference; these operations are denoted as $R_1 \cap R_2$, $R_1 \cup R_2$, and $R_1 \setminus R_2$, respectively. Although not an operation for combining relations, the set operation of negation is also defined for completeness; this operation is denoted as $\sim R$. All of these operations are defined for classes as well as for relations. It is possible for any of these operations to result in a class with no elements or *empty class*, denoted by the symbol \emptyset . Similarly, a *universal class* also exists; denoted by the symbol \mathbf{O} , this class contains all possible data elements.

Methods also exist which restrict relations to contain only specific pairs. A relation may be restricted to particular values in either its domain or in its range by the operations *left-restriction* and *right-restriction*, respectively. Formally, the left restriction operation, expressed as $x(C \rightarrow R)y$, is defined such that given relation R and class C , only those pairs in R are selected whose domain component is also contained in class C .

Similarly, a right restriction operation, expressed as $x(R \leftarrow C)y$, is defined such that only those pairs with range elements contained in C are selected. Simultaneous left and right restriction of a relation is also possible and is expressed as $x(C_1 \rightarrow R \leftarrow C_2)y$. One particular simultaneous restriction, $C \rightarrow R \leftarrow C$, occurs so often in relational programming that the shorthand notation $R \uparrow C$ has been developed for it.

As previously noted, there is no distinction in relational programming between functions and relations – all functions are represented as relations. Unfortunately, the notation xRy is inconvenient to work with when defining a function. To correct this, the alternate notation $y = R(x)$ is used when describing a function; while both expressions are equivalent in definition, the latter is simply easier to deal with. This notation, however, is only valid for one-to-one and many-to-one functions; if multiple result values exist for any input, only one of these values is randomly selected and returned. As demonstrated by the ' $<$ ' relation, many one-to-many and many-to-many functions exist and require the return of all result values. Returning all possible results requires taking the *image* of the input value(s) under the function; this is accomplished by the 'image' relational operator and expressed as **image** $R C$. The 'image' operation selects the image (range) of all values in class C under relation R , deleting duplicate values as required. The *inverse image* of a relation may also be taken; the converse of the image operation, this operation returns those values in the domain of the relation for all values in class C contained in the range of the relation.

Several image related operations also exist. The first is the *unit image*, denoted as **unimg** $R x$, which defines the class resulting from taking the image of individual x under relation R . Similarly, the *inverse unit image* returns that class of elements associated with an individual y in the range of R ; this operation is so common it is referred to as the **all** operation. If the 'all' operation is applied against the '=' relation,

the result is to select the set of numbers equal to x , if they exist; this is referred to as the *unit class operation* and expressed as $\mathbf{un} \ x$. Similarly, the *inverse unit class operation* also exists; this operation, denoted as \mathbf{un}^{-1} , filters out classes which do not consist of a single element.

Up to this point, it has been assumed that all functions are unary; i.e. that they have only one argument. Binary functions, however, are also defined under relational programming. Passing both arguments to the function involved pairing the arguments into a single relation; this relation is then passed as a single input to the function. Binary functions are denoted as $y = f(x,y)$. It is possible to generate a unary function from a binary operation. For example, the binary function $y = x+1$ is denoted as $y = +(x,1)$ when expressed in pure binary notation, however, this may also be expressed as the unary function $y = (+1)x$; the advantage of this second form is that permits complex combinations to be built.

Functions can be combined by the *relative product* operation. Written as $y = f.g(x)$, this operation indicates the serial execution of the functions $u = g(x)$ and $y = f(u)$. Using relations, this operation serves to "cascade" the relations; i.e. the results obtained from the first relation are used as the input to the second relation. Often a relation is combined with itself in order to determine each element's ancestor; for example, given the relation *Parent*, a list of all grandparents can be obtained by the composition *Parent.Parent*. For convenience, a shorthand notation exists for expressing the composition of a relation with itself; this is expressed as R^n , where n indicates the number of compositions of R with itself.

It is also possible to produce a relation, known as an *ancestral relation*, which contains all of the ancestors of an element. Two types of ancestral relations can be

formed, the difference between them being the inclusion of the element itself as an ancestor. The first type of ancestral relation is referred to as an *ancestral of the first kind* ; denoted as R^* , this ancestral represents the reflexive, transitive closure of relation R and is defined as

$$R^* = R^0 \cup R^1 \cup R^2 \cup R^3 \cup R^4 \cup \dots$$

The second type of ancestral relation, referred to as an *ancestral of the second kind*, represents the transitive closure of relation R ; this is denoted as R^+ and defined as

$$R^+ = R^1 \cup R^2 \cup R^3 \cup R^4 \cup R^5 \cup \dots$$

Combinators represent a special set of data manipulation operators in relational programming. While many combinators exist in relational programming, only four are used in this paper; these combinators are:

1. *Parallel* – this combinator, denoted as $R_1 || R_2$, returns all pairs which are in parallel between relations. This operation is defined as

$$(u,v)R_1 || R_2(x,y) = uR_1x \cap vR_2y$$

If the arguments to the parallel combinator are the functions f and g , then the operation $f || g$ is defined as

$$[f || g](x,y) = [f(x),g(y)]$$

which is the element pair formed by $f(x)$ and $g(x)$.

2. *Overlay* – this combinator, denoted as $R_1 ; R_2$, performs the conditional union of relations. This operation is defined as

$$[R_1 ; R_2](x) = R_1(x) \cup [\sim.\text{domain } R_1 \rightarrow R_2](x)$$

which indicates that if x is contained in the domain of R_1 , then $y = R_1(x)$, otherwise $y = R_2(x)$. A common use of the overlay combinator is with the identity function \mathbf{Id} ; this combination extends an operation to always produce a defined result.

3. *Duplicate* – this combinator, denoted as Δ , duplicates relations. This operation is defined as

$$\Delta x = (x,x)$$

where x is the data object to be duplicated, regardless of whether x is an individual, class, or relation.

4. *Duplicate and Parallel* – this combinator, denoted as $f \diamond g$, is a combination of the duplicate and parallel combinators. This operation is defined as

$$(f \diamond g)x = (f \parallel g).\Delta x = (f(x),g(x))$$

Notice that up to this point there has been no discussion of an assignment operator; this is due to the fact that it is rarely used in relational programming. It cannot be eliminated from the language completely, however, since it is occasionally necessary to assign the results of a function to a variable-like object; this is particularly true when defining user-interactive programs where results must be temporarily "memorized" until the user chooses the next function to be executed on the data. Thus, the assignment operator, denoted as $:=$, is defined as

$$R_2 := f(R_1)$$

which indicates that function f is executed on relation R_1 with the results placed in relation variable R_2 .

There also has not been a need as of yet to define specific operators for flow control. Unlike the assignment operator, however, no specific notation need be introduced for flow control; this can be easily handled though the application of existing relational operators. For example, the relational equivalent of the C language if statement is written as

$$C \rightarrow f;g$$

which applies the function f if the input argument is contained in class C ; otherwise function g is applied. Similarly, the equivalent of the **C while** statement is

$$(C \rightarrow f)^* \leftarrow \sim C$$

which continuously applies function f (zero or more times) until the input no longer satisfies C ; output from the process, however, is only permitted when the range of the ancestral relation does not satisfy C , producing the proper result. Changing to an ancestral relation of the second kind produces the equivalent of the **do until** statement, expressed as

$$(C \rightarrow f)^+ \leftarrow \sim C$$

The final topic to be discussed are *records*. The goal is to be able to represent a record, such as a tuple, in a form that is suitable for manipulation by the defined relational operators. A record, however, is nothing more than a finite set of pairs. The first member of each pair represents a unique identifier or *selector* for an associated data element, which constitutes the second member of the pair. Thus, a single record is nothing more than a relation which may be manipulated by any the relational operators already defined. Individual records of same form may be combined into a class of records or *record set*; a record set in relational programming is the equivalent of a relation of degree n in the relational model.

Given a record, a means must be found to manipulate the data contained in that record. If only one field of the record is to be modified, then that can be accomplished directly by the use of the operation $f.R(x)$, where x represents the selector for the field to be operated on. This operation, however, produces only a single result; there are also times when a complete modified record must be produced as a result. The solution is to find a method by where a function can be applied to each field of a record and return a new record as a result. To accomplish this, a type of record known as a *functional record*

is defined; a functional record is a relation where the first element of each pair is a selector and the second element is a function. Thus, a functional record is a "relation of relations". Assuming that there exists a functional record F and a data record R of identical size and with identical selectors, then the operation

$$[F(x)].R(x)$$

produces a partial record result at selector x ; a full result record is produced by the operation $[F].R$.

2.2.2. Example – Implementing Relational Model Operations

In order to demonstrate how relational functions are developed, the relational model operators *Project*, *Select*, and *Join* will be implemented using relational programming operations. This is intended as a brief example in function development; more detailed descriptions of function derivation can be found in [MacLennan82], [MacLennan83], and chapters three and four of this paper.

Throughout this example we assume the existence of two record sets. The first record set, *Addr*, consists of n_1 records, where each record R_1 is of the form $\{name, address, phone\}$. The second record set, *Salary*, consists of n_2 records, where each record R_2 is of the form $\{name, dept, salary\}$.

The first operation, 'Project', is the simplest to implement. For example, projecting out the fields *name* and *address* from *Addr* is accomplished on a per record basis by the left-restriction operation

$$\{name, address\} \rightarrow R_1$$

This operation selects from record R_I only those pairs with the selectors *name* and *address* and produces another record as a result. Extending this projection to all records in the record set involves the application of the **image** operation; this is defined as

$$\text{image } [\{name, address\} \rightarrow] Addr$$

The result of this operation is another record set – the image of *Addr* under the function $[\{name, address\} \rightarrow]$. Generalizing this function completes its definition; thus, the projection operation is defined as

$$\text{Project } attr\ rel = \text{image } [attr \rightarrow] rel$$

The second relational model operation, ‘Select’, chooses records based on the value contained in a particular field of that record. This is a two step process: the first step associates for each record the field in question with the record, while the second step selects records based on the desired field value(s). Examination of the first step reveals that the required function must produce a relation which contains the selected field values in the domain and the record itself in the range; i.e. indexes field values to records. This function is defined as

$$\text{index } attr\ rel = [.attr]^{-1} \leftarrow rel$$

The second step of the function need only produce the image of the values being sought under the indexed record set; this results in a record set containing only the selected records. Thus, selection is defined as

$$\text{Select } attr\ rel\ val = \text{image } [\text{index } attr\ rel] val$$

where the class *val* contains the values being sought. Using the *Addr* relation as an example, the records containing the values *Schaffer* and *Coldwell* in selector *name* are found by the expression

$$\text{Select } name\ Addr \{Schaffer, Coldwell\}$$

The final relational model operation defined is 'Join'. In relational programming, a join is the equivalent of unioning each record in the first record set with each record in the second record set which contains an identical value in the join field. Using the record sets *Addr* and *Salary* as an example, a join on the *name* field is the union of records R_1 and R_2 when the value associated with that selector in R_1 is identical to the value associated with the selector in R_2 . By indexing each record set on the join field, the parallel combinator can be used to match identical domain values. This results in a relation which maps each matched value into a record pair; the first element of the pair consists of record R_1 , while the second element of the pair consists of record R_2 . The final step unions this pair and extends the operation to include the entire record set; thus, the join of *Addr* and *Salary* on the *name* field is defined as

$$\text{Join } name (Addr, Salary) = \text{image} \cup (\text{range} [(index\ name\ Addr) \parallel (index\ name\ Salary)])$$

Generalizing, the 'Join' operation is then

$$\text{Join } attr (rel1, rel2) = \text{image} \cup (\text{range} [(index\ attr\ rel1) \parallel (index\ attr\ rel2)])$$

Chapter Three

Relational Implementation of Data Structures

This chapter shows the relational representation of commonly used data structures and defines the functions which manipulate them. Data structures are the bases of all programs, including operating systems; it is therefore essential that a set of data structures be defined early. The functions which are shown here, or derivations of them, will be used extensively in the next chapter when the relational operating system is defined.

The structures chosen for implementation are the stack, queue, deque, list, array, and tree. One assumption is made in each implementation – only a representation of the data is manipulated and not the data itself. Representations, such as a node identifier, are necessary since the algorithms presented must manipulate unique data elements; user data is rarely unique. Although not shown here, by considering the unique data elements as record selectors, actual data selection can be obtained by extending the given functions with record operations.

Many of the examples presented here were originally presented in [MacLennan82] and [MacLennan83]; the reader should refer to these papers for additional information.

Given the static relational structure of a stack, it is now necessary to define the relational programming equivalents of the stack manipulation operations. The remainder of this section describes how these functions are defined.

The first stack manipulation operation to be implemented determines the depth of the stack; i.e. the number of elements on the stack. Since this is the same as the size of either of the classes which make up the stack relation, the 'depth' function is defined as

$$\mathbf{depth} S = \text{size.domain } S$$

where S represents the *successor* relation describing the stack.

The second stack manipulation operation examines the data element at the top of the stack. To accomplish this, it is necessary to determine which element in the sequence corresponds to the top of the stack. Using the relation presented in figure 3.2 as an example, it can be seen that the top of the stack is the only element which exists in the domain of the relation but not in the range. Given this, a function is defined which produces a resultant class containing only the top of stack element; this function is

$$\mathbf{initial} S = (\text{domain } S) \setminus (\text{range } S)$$

Unfortunately, this function only returns the proper result when applied to a single sequence; if applied to a structure with multiple entry points, the function yields a resultant class which contains every entry point. Insuring the proper result for a stack requires that the resultant class be limited to only a single element, attained by filtering the output of 'initial' through the inverse unit class function. The new function which results from this combination returns the first element of a structure only if that structure has a single entry point; for structures which contain multiple entry points, the function returns \emptyset . This new function is defined as

$$\mathbf{first} S = \text{un}^{-1}.\text{initial } S$$

Now that the location of the top of the stack is known, it is possible to define a function which examines the data element at the top of the stack. This new function, however, needs only to return the set containing the data element at the top of the stack – a task already accomplished by the ‘first’ function. Thus, examining the first data element in a stack is the same as accessing the top of the stack, with the ‘first’ function being used for both.

The third stack manipulation operation deletes the data element at the top of the stack. Deleting that element is the same as returning a list of stack all elements except the one at the top of the stack; the calling program then need only refer to this new relation as the stack. Thus, the deletion function is defined as

$$\mathbf{delete_first} \ S = (\sim.\mathbf{first} \ S) \rightarrow S$$

This function performs satisfactorily under all conditions; a relation S which is either empty or which contains a single element returns \emptyset as a result.

Typically the examine and delete top of stack operations are performed by a single operation known as a "pop". Under relational programming, ‘pop’ is implemented by combining the ‘first’ and the ‘delete_first’ operations such that

$$\mathbf{pop} \ S = \mathbf{first} . (\mathbf{first} \ \diamond \ S := \mathbf{delete_first}) \ S$$

As denoted by the \diamond operator, this function first duplicates S and then performs the ‘first’ and ‘delete_first’ operations in parallel. The ‘delete_first’ function deletes the first element in the sequence and then replaces the contents of relation S ; the ‘first’ function determines the first element of the unmodified S . The result of the combination of the two functions is a relation containing the first element of the stack in the domain and the modified stack relation in the range; the final ‘first’ operation returns the first element from this relation.

The fourth and final data manipulation operation inserts a new data element onto the top of the stack; i.e. a "push" operation. Under relational programming this is functionally the same as creating a new relationship between the element to be inserted and the current top of the stack; this new pair is then made part of the existing relation by the union operation. The newly inserted element automatically becomes the top of stack element since it is now the only element contained in the domain of the relation and not in the range. This function is defined as

$$\text{insert_first } S \text{ elt} = [\text{elt}, (\text{first } S; \{\text{EOF}\})] \cup S$$

where *elt* is the new data element to be linked into the stack. Note that the overlay combinator is necessary only for the case of insertion into an empty stack.

3.2. Queues

The second data structure considered is the *queue*. A queue is similar to a stack in that it is a sequential grouping of data elements with the newest data element at one end of the structure and the oldest data element at the opposite end. The difference between them occurs in accessing the structure; a stack permits data element examination/insertion/deletion only from one end of the structure, while a queue restricts data element examination/deletion to one end of the structure and insertion to the other. Examination and deletion of data elements is performed from the end of the sequence where the oldest (first inserted) data element exists; this is known as the front of the queue. Data element insertion is performed from the end of the sequence where the newest (last inserted) data element exists; this is known as the rear of the queue. Figure 3.3 shows a diagram of a typical queue.

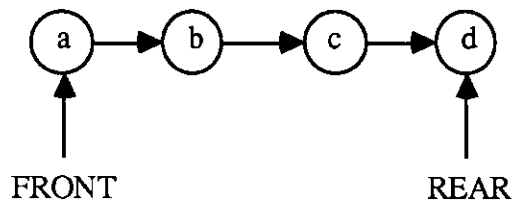


Figure 3.3. A Queue

Given this structure, four types of data manipulation operations can be performed: (1) determine the length of the queue, (2) examine the data element at the front of the queue, (3) delete the data element at the front of the queue, and (4) insert a new data element at the rear of the queue.

As was done with the stack structure, a queue can be relationally implemented by describing the relationship that defines the structure and the operations which may be performed on that structure. Like a stack, the structure of a queue is a sequence of data elements; i.e. the *successor* relation still applies. Given this relational structure, it is now possible to define the data manipulation operations. These operations, however, are quite similar, if not identical, to those already defined for a stack. In only a few instances will extensions need to be made; these are necessary since access to a queue is permitted at both ends of the sequence rather than just the single end defined by a stack.

The first queue manipulation operation determines the length of the queue. This function, however, is the same as the 'depth' function already defined for the stack operations. Since this function yields the length of any sequence regardless of the data model imposed upon it, the function will be retitled 'length' for use by this and all subsequent sequentially based models.

The second and third queue manipulation operations, the examine and delete front of queue operations, are also identical to the equivalent functions previously defined for a stack; examination of the front element of the queue is accomplished by the ‘first’ operation, while deletion of the front element is accomplished by the ‘delete_first’ operation.

In fact, only the last queue manipulation operation, inserting a new data element into the queue, is different than the related stack operation. In this case the function must access the rear of the queue rather than the front, requiring that a function be derived which determines the final element in a sequence. Using the relation in figure 3.2 again as an example, it can be seen that the final element in the sequence is special – it is the only data element associated with the *EOF* marker. This trick makes it easy to quickly find the rear of the queue via a right-restriction; the function which accomplishes this is defined as

$$\mathbf{terminal\ } S = \text{domain } (S \leftarrow \{EOF\})$$

Of course, the same problem exists with the ‘terminal’ function as existed with the ‘initial’ function; given a structure with multiple entry points, the function returns the class containing all of these entry points. The solution, however, is also identical; applying the ‘un⁻¹’ function to the result of ‘terminal’ creates a new function which filters out all multiple entry point structures. Thus, the function which returns the final element of a sequence is defined as

$$\mathbf{final\ } S = \text{un}^{-1}.\text{terminal } S$$

Now that the location of the rear of queue is known it is possible to define the insertion function. As was the case in the stack insert operation, a new relationship must be created between the existing sequence and the new element; in this case, however, that relationship is between the new element and the end of the sequence (without *EOF*). This

is accomplished by the creation of two new pairs in the *successor* relation: the first pair links the final element of the queue with the newly inserted element, while the second pair links the newly inserted element with *EOF*. This new function is defined as

$$\mathbf{ins_final} \ S \ elt = S \cup (\mathit{final} \ S, \ elt \cup (\elt, \{EOF\}))$$

Unfortunately, the ‘ins_final’ function does not properly insert the new element into the sequence; too many links exist, as shown in figure 3.4. The original rear of queue element now points at the newly data element and the *EOF* marker. To correct this problem, the redundant edges in the sequence must be eliminated; a redundant edge is defined as one which links an element and any other element that is not its direct, or first order, successor. By forming a relation which contains only second order (or greater) successors, these may be eliminated from the result relation by the difference operation. Since the second order successors of a relation are obtained by the composition of the relation's ancestral of the second kind and the relation itself, the function to eliminate redundant edges is defined as

$$\mathbf{elim_edges} \ S = S \setminus (S^+ . S)$$

Given this, the insert function is now properly defined as

$$\mathbf{insert_final} \ S \ elt = \mathit{elim_edges} . (\mathit{ins_final} \ S \ elt)$$

This function performs properly under all conditions; insertion in an initially empty queue creates a relation consisting of the entry (elt, EOF) which has no extra edges to be filtered out.

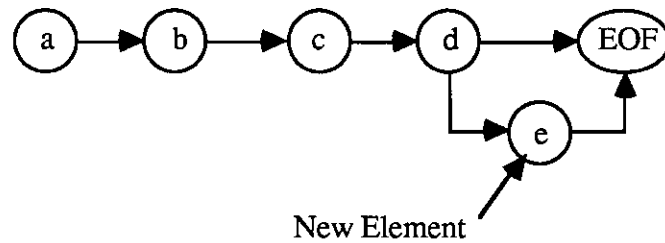


Figure 3.4. Improper Data Insertion in a Queue

3.3. Deques

The third data structure considered is the *deque*. A deque is similar to a queue in that it is also a sequence of data elements; the difference between them occurs in access to the structure. A deque is not restrictive and permits data element examination/insertion/deletion from either end of the sequence. This structure is far more general than the stack or queue and can implement either of the two. In describing the arrangement of the data elements in a deque, the concept of describing element position in the sequence by the age of that element is no longer valid since insertion occurs at either end of the structure. What still holds, however, is that a deque is a sequential structure; this alone is sufficient to relationally describe the arrangement of the data elements in the structure. The ends of the structure are described as being leftmost or rightmost; the choice of which end is considered the left or the right is arbitrary as long as the description is consistent. Given this structure, four data manipulation operations can be performed: (1) determine the length of the deque, (2) examine the data element at either end of the deque, (3) delete the data element at either end of the deque, and (4) insert a new data element at either end of the deque.

In defining the functions for a deque, it is useful to view a deque as a left entry queue combined with a right entry queue. A left entry queue is considered one where new data elements are inserted from the left end of the sequence, while a right entry queue is considered one where data elements are inserted from the right end of the sequence. If the queue structure defined in the previous section is considered a right entry queue, then all of the right entry queue operations have already been defined; further, defining the left entry operations requires that only minor extensions be made to these existing functions.

In fact, the length, examination, and insertion functions are already defined. The 'length' operation applies to any sequence and does not depend on a particular entry point. Examining the data elements on the left and right ends of the deque is accomplished by the 'first' and 'final' functions, respectively. Finally, left end element insertion is defined by the 'insert_first' function, while right end insertion is defined by the 'insert_final' function.

Given this, the only data manipulation operation which needs to be defined is deletion. Deletion is accomplished through two functions, 'delete_first' and 'delete_final'; each operation deletes the data element from the left or right end of the deque, respectively. The function 'delete_first' has already been defined, however, leaving only the 'delete_final' function to be defined. The simplest method of implementing this function would be to restrict the original sequential relation to contain all elements except the one being deleted; unfortunately, this does not work since the process eliminates other data elements as a side effect of the restriction. To prevent this, an ancestral relation of the second kind is formed before the restriction occurs; now after applying the restriction, links exist between all elements except those deleted. This function is defined as

$$\mathbf{del_final\ } S = S^+ \uparrow [\sim.(final\ S)]$$

Note that redundant edges still exist in the result, requiring that the resultant relation be filtered through the 'elim_edges' function; therefore, the final form of the function is defined as

$$\mathbf{delete_final\ } S = \mathbf{elim_edges.del_final\ } S$$

3.4. Lists

The fourth data structure considered is the *list*. A list is a sequence of data elements where access to any element in the sequence is permitted. The list is the most general form of sequential structure and can simulate any of the structures examined so far. Describing the arrangement of data elements in a list is identical to that of the deque with the exception of the ends of the sequence; these are now referred to as the head and the tail of the sequence rather than the left and right. Again, four types of data manipulation operations can be performed on a list: (1) determine the length of the list, (2) examine a data element in the list, (3) delete a data element in the list, and (4) insert a new data element in the list.

Like the other sequentially based structures, the 'length' operation determines the length of a list. Unlike the other sequentially based structures, a list is not restricted to accessing data elements only at the ends of the structures. This necessitates that a method exist which addresses a data element by the position it occupies in the list. In relational programming, this requires converting the data from a relation that links a data element with its successor to one that relates a data element with its index position; this new relation, known as *indexed*, is shown in figure 3.5. The algorithm for forming this relation must determine all of the data elements which are the predecessors of the indexed element; given this, the size of the relation containing all of the predecessors of the

element is the same as that element's index position. The function which accomplishes this is defined as

$$\text{index } S = \text{size.all } S^*$$

The relation produced by this function correlates an element with its index position; index position numbering beginning with one. Two tasks, however, remain before the function is complete. The first task involves eliminating the *EOF* element; this special marker is not needed since all elements of a sequence now exist as an individual entry in the *indexed* relation. The second task switches the columns of the result relation; an indexed list relates index position to data element, not data element to index position. Thus, the final 'index' function is defined as

$$\text{index } S = [\text{size.all } S^*]^{-1} \leftarrow \sim\{EOF\}$$

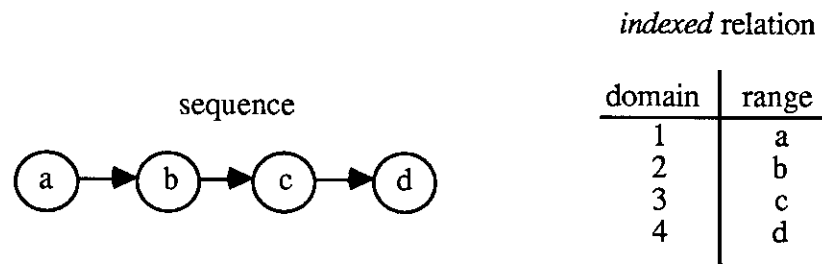


Figure 3.5. Indexed Representation of a Sequence

With the relation converted to a more convenient form, examination of any of the data elements occurs by specifying the indices of interest; the proper elements are then selected by left restriction. This function is defined as

$$\text{element } S \text{ } pos = \text{range } (pos \rightarrow \text{index } S)$$

where *pos* is the class containing the indices of interest. Access to any element position not currently in the list produces a \emptyset result.

The deletion of a data element from a list is only a slightly more complex procedure. Unfortunately, deletion by simple restriction either eliminates necessary data elements or split the list into two separate sequences. Since this problem is the same as the one encountered with the 'delete_final' operation, that function can be used here if generalized to delete any data element in the sequence; this modified function is defined as

$$\mathbf{del_elt} \ S \ elt = \mathbf{elim_edges}.[(S^+) \uparrow (\sim elt)]$$

Deleting an element at a specific position is then

$$\mathbf{delete_element} \ S \ pos = \mathbf{del_elt} \ S \ (\mathbf{element} \ S \ pos)$$

As the function implementing deletion from a list resembles the function 'delete_final', the function which inserts an element into a list resembles the previously defined function 'insert_final'. In this case, however, the function is far more complex; 'insert_final' took advantage of the fact that insertion always occurred in one location, while the list insertion function must be able to insert an element at any location in the sequence. Note that insertion of a new data element into a list is defined as placing that element into the sequence before the specified element; i.e. if the position where insertion will occur is denoted by *pos*, then the new element is inserted before the data element at position *pos*. Appending an element to the end of the list is accomplished by specifying *pos* as *EOF*.

The first task that the list insert function must perform is to determine where to link the new element into the sequence; this requires finding the data element where insertion will occur and its predecessor element. These functions are performed by

$$\mathbf{find_prev} \ S \ pos = [S^{-1} \cdot (\mathbf{element} \ S \ pos)]; \ \mathbf{final} \ S$$

$$\mathbf{find_curr} \ S \ pos = (\mathbf{element} \ S \ pos); \ \{EOF\}$$

The overlay combinator is required in each function to handle the ends of the sequence properly. Note that if the *pos* requested is beyond the end of the list, the functions will

return the position at the end of the list. Given the positions to insert the elements, the elements themselves are inserted via the function

$$\mathit{ins_elt} \ S \ pos \ elt = ([\mathit{find_prev} \ S \ pos], \ elt) \cup (\elt, [\mathit{find_curr} \ S \ pos]) \cup S$$

Eliminating the extra edges completes the function; thus, 'insert_element' is defined as

$$\mathit{insert_element} \ S \ pos \ elt = \mathit{elim_edges}.\mathit{ins_elt} \ S \ pos \ elt$$

3.5. Arrays

The fifth data structure considered is the *array*. Two general types of arrays exist: single-dimensional and multi-dimensional. The single, or one, dimensional array is a list of data elements where all of the elements are related by the type of data they contain; e.g. a list of a store's total sales by month are all members of datatype *dollars*. By definition, the first element of the list is associated with the first index position of the array, the second element with the second index position, etc. From this definition the second and more general multi-dimensional array structure is built. A multi-dimensional array is a grouping of one-dimensional arrays where all of the arrays are of identical length and contain related data elements. Three types of structure manipulation operations can be performed on an array: (1) determine the size of the array, (2) examine a data element in the array, and (3) replace a data element in the array. Insertion and deletion of data elements in an array is not permitted since an array, once defined, is fixed in size.²

The data structure required for a one-dimensional array maps an ordered list of integers to the data element that occupies the index position that the integer represents; i.e. the *indexed* structure defined in the previous section. Thus, a one dimensional array is a list and may be manipulated by the list operations already defined. This can only occur, however, if the data elements can be freely converted between the sequence and array

²While true for most languages, this is not true for all; APL is a notable exception.

structures. The function which converts data from a *successor* structure to an *indexed* structure has already been shown, leaving only the reverse operation to be defined. If we assume that index positions are consecutively numbered in the *indexed* relation, then this new function need only link each element with the element in the index position one greater than the one being examined. After creating this sequential list of elements, the function's final step is to associate the last data element in the list with an *EOF* marker. Appending an *EOF* marker requires two functions; the first function determines which element ends the sequence, while the second function appends the *EOF* marker to that data element. These functions are defined as

$$\mathbf{end\ } S = (\text{range } S) \setminus (\text{domain } S)$$

$$\mathbf{append_eof\ } S = \text{end } S \cup S$$

The final conversion function, 'successor', is then defined as

$$\mathbf{successor\ } I = \text{append_eof}.I.(1+).I^{-1}$$

where *I* represents the *indexed* relation to be converted.

The structure for a multi-dimensional arrays is only slightly more complex. Multi-dimensional arrays require a separate relation for every dimension represented; e.g. a two-dimensional array consists of two relations, presumably *row* and *column*. Each relation is an *indexed* type structure, except that there are now multiple elements at each index position; figure 3.6 shows this structure for a two-dimensional array. Operations on this structure are similar to those already defined – they are simply extended to use multiple relations. For example, assuming a two-dimensional array consisting of relations *row* and *column*, the size of the array is defined as

$$\mathbf{size_array} = ([\text{size.domain } row], [\text{size.domain } column])$$

The function which examines any element in the array is defined as

$$\mathbf{array_elt\ } row\# \ col\# = (\text{image } row \ row\#) \cup (\text{image } column \ col\#)$$

Finally, replacing any element in the array is defined as

$$\begin{aligned} \text{rep_array_elt } \text{row\# } \text{col\# } \text{newdata} = \\ & [\text{row} := (\text{row\#}, \text{newdata}) \cup \text{del_index row row\#}] \parallel \\ & [\text{column} := (\text{col\#}, \text{newdata}) \cup \text{del_index column col\#}] \end{aligned}$$

where

$$\text{del_index } \text{rel } \text{index} = \text{rel} \leftarrow \sim.\text{array_elt row\# col\#}$$

	<i>row</i>	<i>column</i>
	domain	range
	1	a
	2	b
	3	c
	4	d
	5	e
	6	f

	<i>array</i>	<i>column</i>
		domain
		range
	1	a
	2	c
	3	d
	1	b
	2	e
	3	f

Figure 3.6. Relational Representation of a Two-dimensional Array

3.6. Trees

The sixth and last structure considered is the *tree*. A tree represents a significant deviation from the list structures examined so far. A list is a linear structure, so that the successor to any data element is guaranteed to be a set containing at most a single data element. A tree, however, is a non-linear structure; the successor set of any data element can contain multiple elements, provided that all of those elements have only the one predecessor. Certainly a tree is more general than a list; a tree can represent a sequence if all of the sets of successors are limited to one element. This is rarely done, however, and in this paper a list and a tree are treated as two separate entities.

A tree is made up of three types of data elements or *nodes*. The first type of element is a *branch node*, which is a data node associated with a set of successor elements. A branch node is considered the *parent* of each of its successor elements; in turn, each successor is referred to as a *child* of that branch node and a *sibling* of its other children. Children are ordered within a branch node such that the first child is associated with the leftmost branch of parent and the n th child is associated with the rightmost branch. The second type of element in a tree is the *root node*; a root node is defined as a branch node restricted in position to the beginning of the tree. The third and final type of element in a tree is a *leaf node*; a leaf node is exclusively a data node and has no successors.

A tree structure always begins with a single data element, the root node, and its associated set of successors. By definition, a directed arc exists between the root node and each of its successors; each arc is the only path that exists between the root and the child in question. Examining each of the children finds that they are either leaf nodes or branch nodes. If the child is a leaf node, then there are no further children; if the child is a branch node, however, then it has its own set of children with a directed arc to each. This process of descending the tree can continue until there are no further branch nodes to descend; at this point, every node in a tree has been visited and a path from the root established to each.

It is convenient to define levels for describing where a node exists in the tree. If the root of a tree is defined as level zero, then all of the children of the root exist at level one in the tree. Similarly, all of the children of the children of the root exist at level two in the tree. In general, the children of any branch node exist at a level one greater than the level of the branch node. Note that the level in a tree where a node exists is equivalent to the number of arcs traversed in the path from the root to the node in question; this is

referred to as the *path length* to that node. Path length is particularly important when dealing with leaf nodes; for many applications, such as artificial intelligence problems, the choice of the path taken depends on the path lengths involved.

Given this structure, four types of data manipulation operations can be performed on a tree: (1) determine the path length of a particular node from the root node, (2) examine the data associated with a particular node, (3) delete the subtree that begins with a particular node, and (4) insert a subtree in a named branch position of a particular node. In order to minimize the number of functions needed, the definition of a subtree is extended to include trees beginning with a leaf node; this eliminates the need for separate insert and delete operations for branch nodes and for leaf nodes.

Describing a tree via a relation requires that each pair in the relation describe the link between a node and one of its children; thus, for a node with n children, n pairs would exist in the relation. This relation, however, is not sufficient to completely describe a tree; still missing is the information which describes which branch must be taken from a node to get to a particular child. The relational structure required is one which incorporates all of this information, accomplished by having each pair combine a node and branch number to uniquely specify a child. The relation that results from this, *child*, is defined such that

$$x \text{ child } y$$

indicates that y is the child node of x , where x is the *node-branch* pair which leads to y . Figure 3.7 shows a diagram of a typical tree and its equivalent *child* relation.

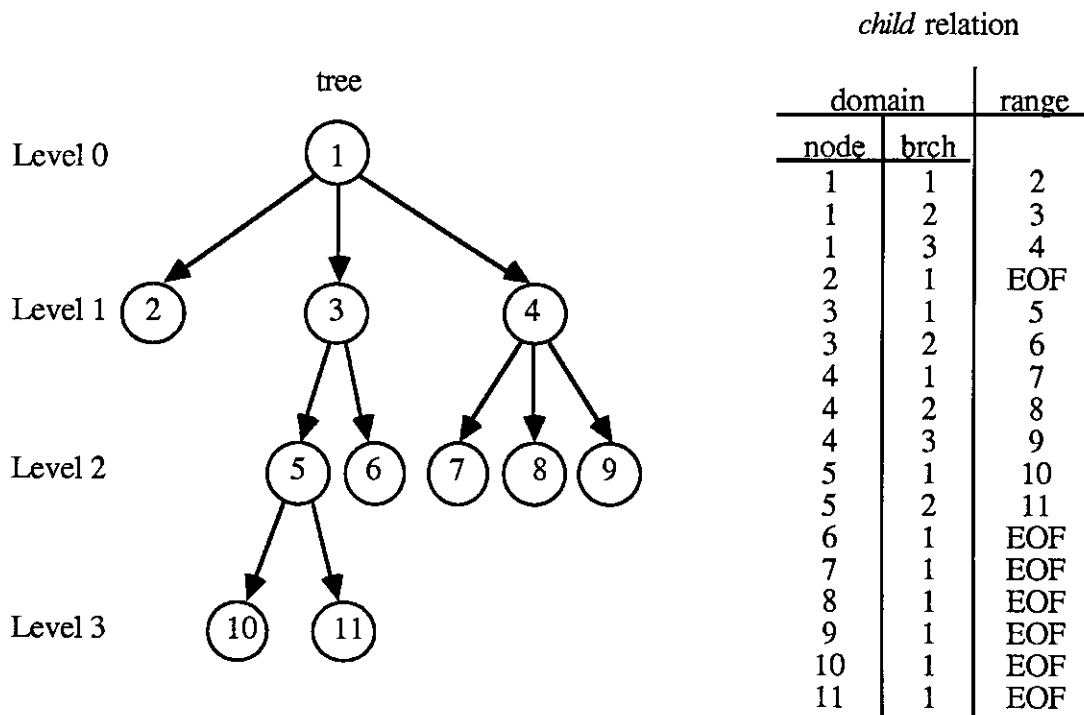


Figure 3.7. Relational Implementation of a Tree

Before defining the data manipulation operations, it is useful to define a set of "utility" functions for trees; these functions, which will be used extensively by the data manipulation operations, determine data element relationships and assist in maneuvering through the tree. These functions are:

1. *Root* – this function yields a class containing the root of the tree. This operation is defined as

$$\mathbf{root } T = \mathbf{first}.\mathbf{(first || Id)}.T$$

2. *Leaves* – this function yields a class containing all of the leaves of the tree. This operation is defined as

$$\mathbf{leaves } T = \mathbf{domain}.\mathbf{terminal } T$$

3. *Descend* – this function descends to the next level of a tree, yielding a child element. This operation is defined as

$$\mathbf{descend } T nd br = [T (nd, br)] \setminus \{EOF\}$$

where T is the relation describing the tree, nd is the node being descended from, and br is the branch of the node to descend. If the specified descent "runs off the end of the tree" (i.e. nd is a leaf node), \emptyset is returned.

4. *Ascend* – this function ascends the tree, yielding the parent element of a node. This operation is defined as

$$\mathbf{ascend } T nd = \text{first}.T^{-1} nd$$

Note that the 'first' operation is required since the inverse of T yields a relation; the 'first' operation yields the parent node. If nd is the root node, the function returns \emptyset .

5. *Left* – this function finds the left sibling of a node. This operation is defined as

$$\mathbf{left } T nd = T.(\text{Id} \parallel (-1)).T^{-1} nd$$

If nd is the leftmost child, the function returns \emptyset .

6. *Right* – this function finds the right sibling of a node. This operation is defined as

$$\mathbf{right } T nd = T.(\text{Id} \parallel (+1)).T^{-1} nd$$

If nd is the rightmost child, the function returns \emptyset .

7. *Children* – this function yields a class containing all of the children of a node.

This operation is defined as

$$\mathbf{children } T nd = \text{image } T (nd \times \text{integer})$$

where *integer* is the class of all integers. Note that the Cartesian product which is formed produces all possible branches from the parent node; if there are no children, the function returns \emptyset .

8. *Siblings* – this function yields a class containing all of the siblings of a node.

This operation is defined as

$$\mathbf{siblings } T nd = \text{children } (T \leftarrow \sim nd).\text{ascend } T nd$$

If there are no siblings, the function returns \emptyset .

9. *Subtree* – this function yields the subset of the tree relation which defines the subtree rooted at a node. This operation is defined as

$$\mathbf{subtree } T nd = [(\times \text{integer}).(\text{children } T nd)^*] \rightarrow T$$

With the utility operations defined, it is now possible to define the data manipulation operations for a tree. The first operation determines the path length to a node. By definition, the path length to a node is equivalent to the level of that node in a tree; thus a function must first be developed which determines the level of each node in the tree. This function, a derivative of the ‘index’ function developed for lists, is defined as

$$\mathbf{level } T = [\text{size}.\text{all } [(\text{first } \parallel \text{Id}). T]^+]^{-1} \leftarrow \sim \{ EOF \}$$

Given this, the path length to of any node in the tree is defined as

$$\mathbf{path_length } T nd = \text{domain } [(\text{level } T) \leftarrow nd]$$

It is also possible to determine the minimum and maximum path lengths to a leaf node are in the tree. This requires that the path lengths to all of the leaf nodes be known; this is accomplished by the function

$$\mathbf{leaf_lengths } T = \text{path_length } T (\text{leaves } T)$$

From this, the functions that determine the minimum and maximum path lengths are

$$\mathbf{min_path_lng } T = \text{first} . < \uparrow . \text{path_leaves } T$$

$$\mathbf{max_path_lng } T = \text{end} . < \uparrow . \text{path_leaves } T$$

Given this, it is also possible to determine which leaf nodes terminate the minimum and maximum paths; this is accomplished by the functions

$$\mathbf{min_path_node } T = \text{image } (\text{level } T) (\text{min_path_lng } T)$$

$$\mathbf{max_path_node } T = \text{image } (\text{level } T) (\text{max_path_lng } T)$$

The result of these functions is the class containing the leaf nodes which are on the minimum/maximum path(s), permitting direct access to these nodes. If desired, the minimum/maximum path itself may then be traversed simply by ascending the tree until the root node is reached; if a descending traversal is also required, a stack of nodes can be formed during the ascent for use later.

The second data manipulation operation examines the data associated with a node. As was true in the previous structures studied, this is same as returning the node itself, accomplished via the tree traversal operations already described.

The third data manipulation operation deletes a subtree. This is accomplished by (1) removing all of the *EOF* markers, (2) determining the nodes in the subtree to be deleted, (3) restricting those nodes from the tree relation, and (4) replacing the *EOF* markers on the leaves of the tree. The first part of the function determines which nodes are to be deleted and deletes them; this is defined by

$$\mathbf{del_tree } T nd = T \uparrow [\sim .(\text{children } T nd)^*]$$

The remainder of the function manipulates the *EOF* markers; this is defined as

$$\mathbf{delete_tree } T nd = \\ [\mathbf{del_tree } (T \leftarrow \sim \{EOF\}) nd] \cup [(\times \{EOF\} . \text{end} . \mathbf{del_tree } (T \leftarrow \sim \{EOF\}) nd)]$$

The final data manipulation operation inserts a subtree at a specified node and branch; if a subtree already exists at the specified branch, it is deleted. Given this, the insertion operation is defined as

$$\mathbf{insert_tree } T nd br new = [(T^{-1} nd), (\text{first} . \text{first } new)] \cup [\mathbf{del_tree } T nd]$$

where *new* specifies the relation for the new tree to be inserted.

Chapter Four

Operating System Implementation

This chapter considers various operating systems concepts and their implementation using relational programming. In examining each concept, no attempt is made to correct any inefficiencies or problems inherent to that algorithm; analysis and modification of each algorithm is beyond the scope of this paper. Finally, several performance related assumptions are made throughout the chapter; for example, disk access is assumed to be instantaneous. These assumptions do not change the basic nature of each implementation and may be eliminated by extending the defined functions.

4.1. Physical Devices

Every computer system consists of a variety of physical devices; it is the task of the operating system to supervise these devices, regulating their allocation and use. In addition, the operating system provides a medium through which programs in the system view devices. One trend in operating systems design is to represent devices in a manner identical to other system structures; for example, in the Unix¹ operating system each device is viewed as if it were a file [Ritchie74]. This paper uses a similar concept; the

¹Unix is a trademark of ATT Information Systems.

difference is that physical devices are mapped onto relations rather than files. Though many types of devices exist, only three will be considered here: the central processing unit (CPU), primary storage (main memory), and secondary storage (disk). A description of each of the devices is presented below, with diagrams of the associated structures shown in figure 4.1.



Figure 4.1. Physical Device Relations

The CPU, the heart of the computer system, interprets program instructions and produces the desired results. Associated with the CPU is a high speed memory set which denotes its current state; this state information includes the current instruction being executed, the address of the next instruction to be executed, status flags, etc. In most computer systems this high speed memory is represented as a set of registers. This paper takes a slightly different approach; here the CPU state information is assumed to be entirely contained in a single record of the relation *CPU*. Each attribute of this relation is the equivalent of a hardware register; i.e. they represent the instruction register, program counter, etc. Operations performed on this relation produce the expected results; reading an attribute returns the current state of that register while writing to an attribute modifies the register, changing the system state.

The second major device to be mapped to a relation is main memory. Memory is comprised of two distinct parts – a physical address and the data contained at that address. Thus, the equivalent relation *memory* must consist of *address-data* pairs, one for each byte in main memory. Reading or writing data to this relation is the functional equivalent of reading/writing the data to main memory at the specified addresses. An identical structure is used for mapping the third major device, *disk*. Note however, that the domain of *disk* could easily be redefined to include physical device information (i.e. disk number, cylinder, sector) rather than just the simple byte address used here.

4.2. Processes

The concept of a *process* is truly central to an operating system, for the operating system itself is a process – the process which manages the system. For the purposes of this discussion a process refers to any program currently in a *state of execution*, where a state of execution is defined as either: (1) *running*, i.e. the program is currently executing instructions on a CPU, (2) *ready*, i.e. the program is ready to run but is waiting its turn for a CPU, or (3) *blocked*, i.e. the program is waiting for the completion of an external event before it is ready to compete for CPU time.

Associated with each process is a unique identifier known as its *pid*. Identifiers are drawn from the class *pids*; this class, a subset of the class *integer*, contains all of the valid process identifiers in the system.² Process identifiers are used to represent each process in the various operating system structures; these structures record the priority of the process in the system, the memory assigned to the process, etc. One such structure is the record set *pstate*; each record in this set is identical in structure to the record *CPU*,

²Unique identifier sets are subsets of the class *integer* simply for convenience; integers are easy to manipulate. For the remainder of this chapter, therefore, any class of unique identifiers defined for a structure will be a subset of *integer*.

thus serving to describe the state of the system as of the last time the process executed. By saving the current state of the CPU in *pstate* and loading *CPU* with the state record of another process, a *context switch* occurs; this is assumed to be a machine level operation invoked by calling `context_switch` with the *pid* of the new process to be executed. A second assumed function, `cur_pid`, returns the *pid* of the process currently executing.

4.2.1. Process Creation

Processes are created as the result of an explicit request by the currently executing process. When this occurs, a *parent-child* relationship is established between the creating and created processes. A relation, known as *ptree*, is used to record this hierarchy; this relation is shown in figure 4.2. *Ptree* serves serves two purposes in the operating system; in addition to recording the process hierarchies, it also indicates all of the active processes in the system.

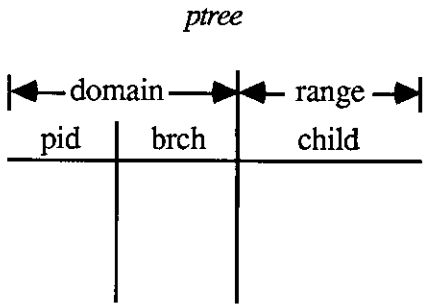


Figure 4.2. Process Hierarchy Relation

The first step in creating a new process obtains an unused process identifier. Since no explicit list is kept of unused identifiers, it is necessary to calculate them by determining the difference between the class containing all of the identifiers and the class containing those identifiers currently in use; any element of the class returned by this

function can be chosen as the new process identifier. Given this, the function which returns the first free process identifier is defined as

$$\mathbf{new_pid} = \text{lo.}[pids \setminus (\text{domain.domain } ptree)]$$

where

$$\text{lo } \mathbf{class} = \text{first.sort } \mathbf{class}$$

$$\text{sort } \mathbf{class} = \langle \uparrow.(\{EOF\} \cup \mathbf{class})$$

The second step in creating a process builds the process's CPU state record. Defining the function which creates this record, however, is difficult since the complete structure of *pstate* is unknown. Thus, we assume the existence of the function **create_pstate** which, given the *pid* of the new process, creates an initial state record for that process.

The third and final step in creating a process establishes the relationship between the new process and its parent; this operation, however, is simply an application of the 'insert_tree' function defined previously. Given this, the function which creates a new child process for *pid* is defined as

$$\mathbf{create_process} = \text{first.}(\text{Id}\hat{\Delta}[\mathbf{c_state}\hat{\Delta}\text{ins_ptree}]).\mathbf{new_pid}$$

where

$$\mathbf{c_state } \mathbf{pid} = \mathbf{pstate} := \text{create_pstate } \mathbf{pid}$$

$$\mathbf{ins_ptree } \mathbf{pid} = \mathbf{ptree} := \text{insert_tree } \mathbf{ptree} \text{ cur_pid } \mathbf{new_brch} (\mathbf{pid}, \{EOF\})$$

$$\mathbf{new_brch} = (+1).\text{size.children } \mathbf{ptree} \text{ cur_pid}$$

Deleting a process is the reverse of creating one with one modification; when a process is deleted, all of its children are immediately terminated. Since this is just an application of the 'delete_tree' function, the process deletion function is defined as

$$\mathbf{delete_process } \mathbf{pid} = [\text{delete_pstate}\hat{\Delta}(\mathbf{ptree} := \text{delete_tree } \mathbf{ptree})] \mathbf{pid}$$

where `delete_pstate` is the function which deletes the CPU state records for the terminated processes. Note that no explicit function is required to release the process identifiers; this occurs automatically by their not being included in the *ptree* relation.

Unfortunately, the functions described here for process creation and deletion are incomplete; missing are the operations which insert and delete the process from other system structures (such as the scheduling queue). Since these operations will not be defined until later in the chapter, however, it is simply noted that the 'create_process' and 'delete_process' functions should be expanded to include them.

4.2.2. Process Interaction

For the most part, processes are independent of each other, unconcerned with the actions of the other processes in the system. Situations do arise, however, when two (or more) processes must interact; this interaction may be in the form of *mutual exclusion*, *process synchronization*, or *process communication*. Mutual exclusion algorithms prevent concurrent access to a limited or shared resource by different processes; these algorithms serialize the execution of the processes. The execution order of the processes is irrelevant as long as they execute in disjoint timeframes. When order is important, process synchronization algorithms must be employed; these algorithms synchronize processes by delaying the execution of the first until the second has completed a mutually agreed upon milestone or *event*. Finally, process communication algorithms must be employed when the limited form of communication provided by mutual exclusion and process synchronization proves insufficient for the information exchange required between processes.

4.2.2.1. Signals

The first process interaction mechanism to be examined is a simple signalling scheme used for process synchronization. Under this scheme, a process P_1 , which is dependant upon a second process P_2 , suspends execution (i.e. blocks) until notified by P_2 that the mutually agreed upon event has been completed. The process which blocks is referred to as *waiting* on the event; the process which notifies the blocked process of event completion is referred to as *signalling* that event.

While any number of processes can wait on the same event, the question is how many of these processes should be notified when the event completes. There are two general solutions to the problem: either (1) only one of the waiting processes is notified, or (2) all of the waiting processes are notified. Under the first approach, waiting processes form a FIFO queue on each event; when that signal arrives the first process in the queue is notified and unblocked. Under the second approach, no queue exists; all processes waiting on an event are unblocked when the signal is sent. This paper uses both approaches; signal implementation takes the approach of notifying all processes, while the FIFO queue approach will be discussed in section 4.2.2.2.

Signal implementation under relational programming requires a data structure which relates processes with the events they are waiting on; this relation, known as *waiting*, is shown in figure 4.3. The operations required to manipulate this relation are minimal, for their only function is to add and delete *pid-event* pairs. Process blocking is implicit, accomplished simply by the process's existence in the *waiting* relation. Thus, the signalling functions are defined as

$$\mathbf{wait\ event} = \mathbf{run_next.}[waiting := waiting \cup (event \times cur_pid)]$$
$$\mathbf{signal\ event} = \mathbf{waiting := (\sim event) \rightarrow waiting}$$

where *event* represents the class of valid events on which the process is waiting/signalling and 'run_next' schedules another process for execution; see section 4.2.3 for a full description of the 'run_next' function.

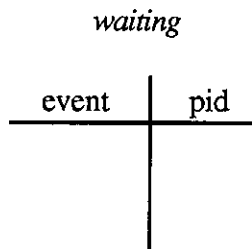


Figure 4.3. Signalling Relation

4.2.2.2. Semaphores

Another method of process synchronization is provided by Dijkstra's *semaphores* [Dijkstra65]. Semaphores correct a basic problem encountered with signals – signalling systems have no memory, so that a signal sent with no process waiting is lost. A semaphore, on the other hand, is a form of history mechanism, retaining the knowledge that an event occurred which no process saw. Two types of semaphores exist: *binary semaphores* and *counting semaphores*. Binary semaphores simply remember that an event occurred, regardless of how often that event occurred. Counting semaphores, on the other hand, retain a count of how often each event occurred; in this section only counting semaphores will be considered.

Two relations are required to implement semaphores. The first relation, known as *semaphores*, contains each semaphore and its associated event count. The second relation, known as *waiting*, associates each semaphore with the queue of blocked

processes waiting on it; this is essentially the same structure required to implement the FIFO queue approach to signal reception. Figure 4.4 shows a diagram of both relations.

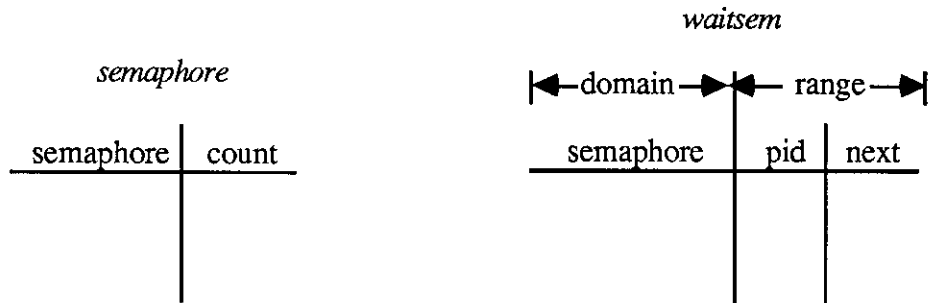


Figure 4.4. Semaphore Relations

Both relations are manipulated by the semaphore operations P and V , which can be considered expanded versions of their signalling counterparts ‘wait’ and ‘signal’. In addition to adding and deleting processes, however, these operations must also account for the current value of the semaphore and manipulate that value as required. Given this, the P operation is defined so that the calling process will wait on a semaphore only if the value of that semaphore is zero; otherwise the process will proceed and the value of the semaphore decremented by one. Similarly, the V operation is defined so that when it is called the first process waiting on that semaphore is permitted to proceed; if no process is waiting the value of the semaphore is incremented by one. In implementing these functions we assume the existence of a class *pos_int* which consists of all positive integers including zero; given this, the semaphore functions are defined as

$$P \text{ sem} = (\text{decr sem}) \parallel (\text{wait_sem.domain.sem} \rightarrow \text{semaphore} \leftarrow \{0\})$$

$$V \text{ sem} = [\text{incr.(sem \setminus) \diamond \text{sig_sem}}].\text{domain (sem} \rightarrow \text{waitsem)}$$

where

$$\text{decr sem} = \text{semaphore} := ([(\text{Id} \parallel (-1)).(\text{sem} \rightarrow \text{semaphore})] \leftarrow \text{pos_int}); \text{semaphore}$$

```

incr sem = semaphore := [(Id || (+1)).(sem → semaphore)]; semaphore
wait_sem sem = waitsem :=
    (sem ×.insert_final cur_pid [image waitsem sem]); waitsem
sig_sem sem = waitsem :=
    waitsem ← [del_elt (range waitsem) (first.image waitsem sem)]

```

4.2.2.3. Mailboxes

Mailboxes provide a method of general communication between processes. This scheme assigns each process a unique mailbox; when a message is sent by another process, it is delivered to this mailbox and held until collected by the mailbox owner. Should the owner request delivery of a message before it has been posted, the process blocks until the message is sent; i.e. the reading and writing processes are synchronized. Process synchronization can be accomplished by any of the methods already discussed; for this example signals are chosen.

Only one relation needs to be defined for mailbox implementation. This relation, known as *mail*, associates process identifiers with their mailboxes; a diagram of this relation is shown in figure 4.5. Two operations manipulate this relation: the first posts messages to a mailbox while the second retrieves the posted messages. These functions are defined as

```

send_msg pid_msg = [signal.domain∧(mail := mail ∪)] pid_msg
get_msg = cur_pid → mail; [get_msg.wait cur_pid]

```

where *pid_msg* consists of the pair *pid-message*; this form is used so that multiple messages may be sent with one call.

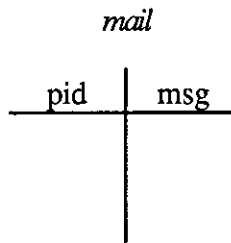


Figure 4.5. Mailbox Relation

4.2.3. Process Scheduling

The scheduling process allocates the primary system resource – the CPU. Scheduling algorithms are classified as either *preemptive* or *non-preemptive*. In a preemptive algorithm, a running process receives exclusive CPU use for a finite amount of time or *quanta*; when that quanta expires the process is interrupted and another process selected for execution. Once interrupted, the process must wait until there are no higher priority processes in the system before it again receives CPU service. Non-preemptive algorithms, on the other hand, do not interrupt the process during its CPU allocation; processes run until completion. In a typical time-sharing system, the job blend will consist mostly of short, interactive type processes with only a few processor intensive, batch type processes seen; non-preemptive algorithms prove a poor choice in this type of system because of the excessive delays that occur when a batch process executes. In this paper only preemptive algorithms will be examined.

4.2.3.1. Round-Robin

In round-robin scheduling a single FIFO queue exists from which processes are selected for execution. When a new process enters the system, it is given the lowest priority assignment in the system – at the rear of the queue. As time is spent in the system the process gains priority, advancing towards the front of the queue until it is finally the highest priority process in the system. When a CPU next becomes free, this process is selected for service and executes for a maximum of one quanta. If the process completes during this time, it exits the system; if not, it is cycled back to the end of the queue to await additional CPU time. It can be seen that this scheme provides equal service to both interactive and batch processes.

A relational programming implementation of round-robin scheduling requires only one data structure; this structure, shown in figure 4.6(a), forms the FIFO queue of process identifiers. Two operations can be performed on this structure: adding a process to the queue and selecting the highest priority process for execution. At first glance, these operations appear to be just applications of the standard queue operations. In fact, the first of these functions is such an application; this function, which appends a process to the queue, is defined as

$$\text{schedule } pid = sched := \text{insert_final } sched \text{ } pid$$

The second function, however, is not accomplished by simply applying an existing function; consideration must be made for blocked processes. To account for this, the selection operation must be redefined so that the process selected for service is the highest priority *ready to run* process in the system. Given this change, the function which selects the highest priority function is defined as

$$\text{select_process} = \text{first}(\text{Id} \diamond \text{pop_proc}).\text{first}.\text{elim_blocked}$$

where

pop_proc *pid* = *sched* := del_elt *sched* *pid*

elim_blocked = del_elt *sched* (~.range *waiting*)

In this case the only blocked processes are those waiting on a signal, however the function may be easily modified to include other blocking mechanisms. Finally, executing the selected process involves a simple context switch; therefore, the function which schedules a new process is defined as

run_next = context_switch.select_process

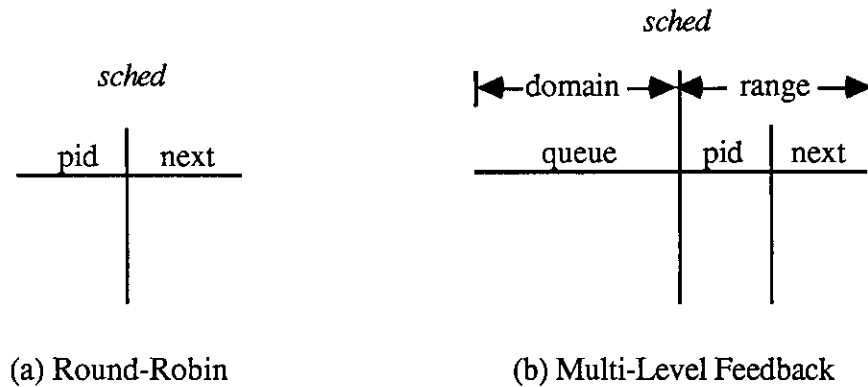


Figure 4.6. Process Scheduling Relations

4.2.3.2. Multi-Level Feedback

Multi-level feedback scheduling can be considered a variation of the round-robin algorithm. One problem with the round-robin scheme is that there is no distinction between interactive and batch processes; interactive processes simply complete in fewer queue cycles. If batch processes are considered to be of low priority in the system, however, then each batch process in the queue only serves to delay CPU service for the high priority interactive processes. Multi-level feedback algorithms correct this deficiency by penalizing processes which require large amounts of CPU time, i.e. batch processes.

In a multi-level feedback scheme, there exists not one but k process queues (Q_1, Q_2, \dots, Q_k), each with an associated priority level (P_1, P_2, \dots, P_k , where $P_1 > P_2 > \dots > P_k$). The next process selected for CPU service always resides in the highest priority non-empty queue; thus, for a process residing in the k th queue to receive service, queues 1 through $k-1$ must be empty. When a new process enters the system it is enqueued on Q_1 regardless of whether it is an interactive or batch type process.³ If all processes are runnable, then the next to receive CPU service will be selected from Q_1 . If this process completes during its allocated quanta, it exits the system; if not, it is lowered in priority and enqueued on Q_2 . At this lowered priority, the process will not be selected again for CPU service until it is the highest priority process in Q_2 and Q_1 is empty. When finally selected, if the process still does not complete during its quanta it again drops in priority and is enqueued on Q_3 . Thus, assuming that a process has not run to completion, it will always reside in queue $q+1$, where q is the number of quanta the process has already received. Note that while this type of algorithm provides for fast response to interactive processes, it accomplishes this at the expense of batch processes; batch processes can now experience excessively long delays before receiving CPU service.

Certainly a method of implementing the multiple queues required by this algorithm is to provide k relations, one per queue. A far simpler approach, however, combines the queues into a single relation; this is possible since the queues consist of disjoint sets of elements. Queue identifiers must also be incorporated into this relation, for without them it would be impossible to know which queue has higher priority. Figure 4.6(b) shows the final structure of this relation. Multiple queues make the functions which implement multi-level feedback scheduling more complex than the equivalent round-robin functions;

³Generally this cannot be predetermined.

the intent of each function, however, is the same. Given this, the scheduling functions are defined as

schedule *pid queue = sched := (queue ×.insert_final pid [image sched queue]); sched*
select_process = first.(Id∧pop_proc).first.image.(Id∧lo.domain).elim_blocked

where

pop_proc *pid = sched := sched ← [del_elt (range sched) pid]*
elim_blocked = *sched ← [del_elt (range sched) (~.range waiting)]*

4.3. Virtual Memory

Virtual memory systems provide each process with a *virtual address space* much larger than the available main memory. To accomplish this, only small sections of a process reside in memory at any one time; each section is referred to as a *page*. It is the responsibility of the system to map any reference to a virtual address by the process into the appropriate physical memory address. Referencing an address not currently in main memory results in a *page fault*, a process which causes the system to automatically load the proper page from secondary storage. To accommodate pages, memory is partitioned into equal size sections known as *page frames*; each frame is the same size as one page and is used as a container for a page when it is in main memory.

Implementing virtual memory requires defining several new data structures. The first of these structures is a list of all of the possible virtual pages; this class, known as *vpages*, is a subset of the class *integer*. The next structures to be defined, *frames* and *pages*, relate each frame/page with the set of addresses on primary/secondary storage that it occupies. The last structures to be defined are the relations *p_frames* and *p_pages*; these relations map process virtual page references to a physical frame/page. A diagram of each of the relations is shown in figure 4.7.

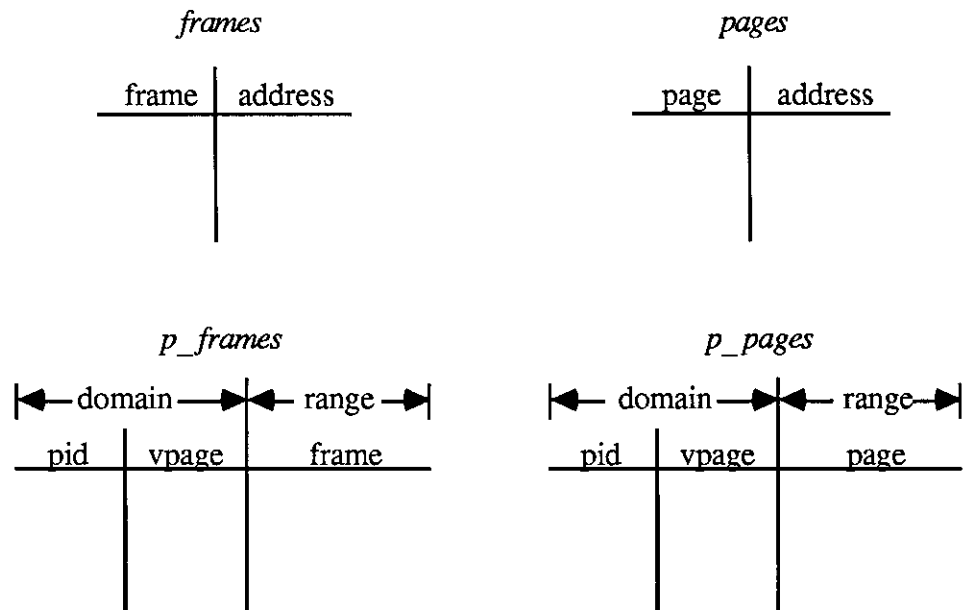


Figure 4.7. Virtual Memory Relations

4.3.1. Memory Allocation

The first step in allocating memory to a process determines the number of pages requested. Memory is only allocated in pages; requests which are not a multiple of the page size are rounded upward to the next multiple. This is accomplished by the function

$$\text{round_up } req = \text{lo.domain}.[(\text{Id} \parallel (*\text{pg_size})).\text{integer}] \leftarrow (\text{all} \geq req)]$$

where

$$\text{pg_size} = \text{size.lo.domain.frames}$$

and * represents the multiplication function. Once this is determined, a set of continuous virtual pages equal in length to the request is allocated in the process's virtual address space; when additional memory is requested in a virtual system, the allocation occurs only in the virtual address space of the requesting process. This function is defined as

$$\mathbf{alloc_vpage} \mathit{req} = \mathbf{extend} \mathit{req} (\mathbf{first_fit}.\mathbf{round_up} \mathit{req})$$

where

$$\mathbf{first_fit} \mathit{req} = \mathbf{lo}.\mathbf{image} [\mathbf{seq_size}].\mathbf{all} \geq \mathit{req}$$

$$\mathbf{seq_size} = \mathbf{index}.\mathbf{[elim_edges.sort.vpages \setminus unimg} (\mathbf{domain} \mathit{p_pages}).\mathbf{cur_pid}]^{-1}}$$

$$\mathbf{extend} \mathit{num} \mathit{base} = [\mathbf{all} \geq \mathit{base}] \cap [\mathbf{all} < (\mathit{base} + \mathit{num})]$$

Next, each virtual page is associated with a physical page in secondary storage; this is accomplished by the function

$$\mathbf{alloc_page} \mathit{vpage} = \mathbf{map_addr} \mathit{vpage}.\mathbf{[domain} \mathit{pages} \setminus \mathbf{range} \mathit{p_pages}]}$$

where

$$\mathbf{map_addr} \mathit{s_addr} \mathit{d_addr} = [\mathbf{mk_map} \mathit{d_addr}].\mathbf{all} [\mathbf{mk_map} \mathit{s_addr}]$$

$$\mathbf{mk_map} \mathit{addr} = \mathbf{index}.\mathbf{elim_edges.sort} \mathit{addr}$$

The result of this function is used to update the $\mathit{p_pages}$ relation; the function which accomplishes this is defined as

$$\mathbf{up_pgrel} \mathit{pgrel} = \mathit{p_pages} := ([(\mathbf{cur_pid} \times) \parallel \mathbf{Id}].\mathit{pgrel}); \mathit{p_pages}$$

The final function is the actual entry point to the memory allocation operation, tying all of the previous functions together as one. This function is defined as

$$\mathbf{alloc_mem} \mathit{req} = \mathbf{first}.\mathbf{[lo} \hat{\Delta} (\mathbf{up_pgrel}.\mathbf{alloc_page})].\mathbf{alloc_vpage}.\mathbf{round_up} \mathit{req}$$

This function returns the first address of the allocated memory in the virtual address space.

Releasing a virtual page from the address space only requires removing the proper vpage entry from $\mathit{p_frames}$ and $\mathit{p_pages}$; this function is defined as

$$\mathbf{rel_page} \mathit{vpage} = [\mathbf{del_frames} \hat{\Delta} \mathbf{del_pages}].\mathit{vpage}$$

where

$$\mathbf{del_frames} \mathit{vpage} = \mathit{p_frames} := (\sim.\times \mathit{vpage}.\mathbf{cur_pid}) \rightarrow \mathit{p_frames}$$

$$\mathbf{del_pages} \mathit{vpage} = \mathit{p_pages} := (\sim.\times \mathit{vpage}.\mathbf{cur_pid}) \rightarrow \mathit{p_pages}$$

Notice that if the page is shared between processes it is only deleted from the address space of the requestor; while it is still in use, it is not physically deleted from either primary or secondary storage.

4.3.2. Paging

When a page fault occurs, the operating system must load the referenced virtual page into main memory. To accomplish this, the operating system must decide which frame to be allocate to the page. If unallocated frames exist, the decision is simple – any one of the free frames is chosen. When the pool of available memory has been exhausted, however, a page frame must be freed so that execution can continue. The strategy for choosing which frame will be freed is referred to as the *replacement policy* of the system; specific replacement policies are discussed in section 4.3.3.

The first step in implementing the page fault mechanism is to allocate a frame in main memory for the page being brought in; this is accomplished by the function

alloc_frame = lo.domain.[(domain *frames* \ range *p_frames*) → *frames*; *rpl_frame*]

The first free frame is chosen if one exists; if not, the page replacement function ‘*rpl_frame*’ is called to free one of the frames. Once this is done, a new *vpage-frame* mapping must be created for the process and the *p_frames* relation updated with this information; this is accomplished by

up_frmrel *vpage frame* = *p_frames* := [*cur_pid*, *vpage*), *frame*]; *p_frames*

Next the page must be copied from secondary to primary storage, mapping the addresses appropriately; this is done by

cp_mem *vpage frame* = *memory* :=

[(map_addr (image *frames frame*)) || Id].[(disk_addr *vpage*) → *disk*]; *memory*

where

$$\mathbf{disk_addr\ vpage} = \mathbf{image\ pages.p_pages} (\mathbf{cur_pid} \times \mathbf{vpage})$$

The final function to be defined merges all of the previous operations into one; thus, the function which handles page faults is defined as

$$\mathbf{fault\ vpage} = [\mathbf{cp_mem\uparrow\ frmrel}].\mathbf{vpage} \times .\mathbf{lo.alloc_frame}$$

Shared memory is not supported in this version of the 'fault' function. Should shared memory be desired, the only modification required is to check if the virtual page has already been loaded by another process before attempting to do so.

4.3.3. Page Replacement

When a page fault cannot be satisfied due to a lack of available memory, the page replacement algorithm selects an allocated frame and frees it. Ideally, the frame freed is the one which will not be referenced for the furthest time into the future; this choice yields the best possible system performance [Belady66]. Unfortunately, absolute knowledge of which frame this is requires a crystal ball – a hardware device not yet well defined. Lacking this, various heuristics have been devised to predict which frame is the least likely to be referenced. Although several strategies exist, only one is described here – the Least Recently Used (LRU) algorithm.

Under a LRU replacement strategy, the page frame which has not been referenced for the longest period of time is selected for replacement; the assumption is that this frame has the lowest probability of being referenced in the near future. Implementation of this algorithm requires that every page frame be associated with a timestamp; when the frame is referenced, its timestamp is updated. At page replacement time, the frame with the oldest timestamp is selected from a set of candidates and freed. This set of candidate frames may be drawn either from local memory or global memory. If drawn from local

memory, only those page frames occupied by the faulting process are considered for replacement. If candidates are drawn from global memory, then all page frames in memory are considered in the replacement decision without regard to which process occupies them; in this paper candidate page frames will be drawn from global memory.

Implementation of the LRU strategy requires two data structures. The first structure, known as *timestamp*, relates each frame in memory with its current timestamp; a diagram of this structure is shown in figure 4.8. The second structure, known as *dirty*, is a class of page frames identifying those which have been modified but not written back to disk; it is preferable not to use these frames as candidates since their secondary storage copy must be updated before they can be freed. For efficiency, the hardware updates both data structures directly whenever a page frame is referenced.

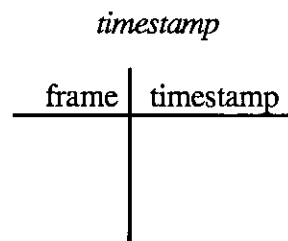


Figure 4.8. LRU Relation

The function which implements the LRU strategy first attempts to select the oldest clean frame in the system for replacement; if none exist the oldest dirty frame is chosen and written out to secondary storage. In either case, the *pframe* relation is updated so that no process owns the newly freed frame; references to the discarded page will cause a page fault. Given this, the LRU replacement function is defined as

$$\mathbf{rpl_frame} = \text{first}.[\text{Id} \diamond (\text{write_dirty} \diamond \text{elim_frm})].\text{sel_rpl}$$

where

$$\begin{aligned}
\text{sel_rpl} &= \text{lo.image (candidates)}^{-1}.\text{lo.range candidates} \\
\text{candidates} &= [\text{domain}.\text{((}\sim\text{dirty)}\rightarrow\text{frames}; \text{frames})]\rightarrow\text{timestamp} \\
\text{write_dirty frame} &= \text{cp_disk}.\text{[(range.image (p_frames)}^{-1})\diamond\text{Id}].(\text{frame} \cap \text{dirty}) \\
\text{cp_disk vpage frame} &= \text{disk} := \\
&[(\text{map_addr (disk_addr vpage)}) \parallel \text{Id}].[(\text{image frames frame}) \rightarrow \text{memory}]; \text{disk} \\
\text{elim_frm frame} &= \text{pframes} := \text{p_frames}\leftarrow(\sim\text{frame})
\end{aligned}$$

4.4. Filesystems

The *filesystem* manages the secondary storage area, providing a permanent storage location for programs and data. Some of the tasks performed by the filesystem are:

1. managing secondary storage allocation.
2. organizing files into manageable structures.
3. mapping from a representation of the data, i.e. a file name, to the data itself.
4. providing a security system for files.
5. reading and writing files.

Most of these operations are variations of previously defined functions and will not be repeated here. Only one function will be defined here, that which maps filenames into actual files (*open*); the mechanism for reading or writing the file itself is an application of the paging mechanism.

Filesystems are usually organized into either *flat* and *hierarchical* structures. In a flat file structure, all files in the filesystem are organized on one level. There are two disadvantages with this type of organization: (1) if many files exist, it may be difficult to find the one of interest, and (2) all file names used in the filesystem must be unique. In a hierarchical organization, files reside within *directories* which break the filesystem into

smaller, more manageable pieces. The advantage of this is that a filename need only be unique within a directory rather than across the filesystem. The disadvantage of this type of structure is that a user must know exactly where the file is located in the structure; if the tree is very complex, finding a file in an unknown location is difficult. Since a flat filesystem can always be implemented under a hierarchical one, a hierarchical implementation will be discussed here.

Three relations are necessary to implement a hierarchical filesystem. The first relation, known as *dtree*, represents the file hierarchy. In this tree, branch nodes indicate directories and leaf nodes indicate files. The second relation, known as *filenames*, maps unique *dtree* node identifiers to external filenames. The third relation, known as *files*, maps the same node identifiers to the sequence of pages in secondary storage which make up the actual file. Figure 4.9 shows a diagram of these relations.

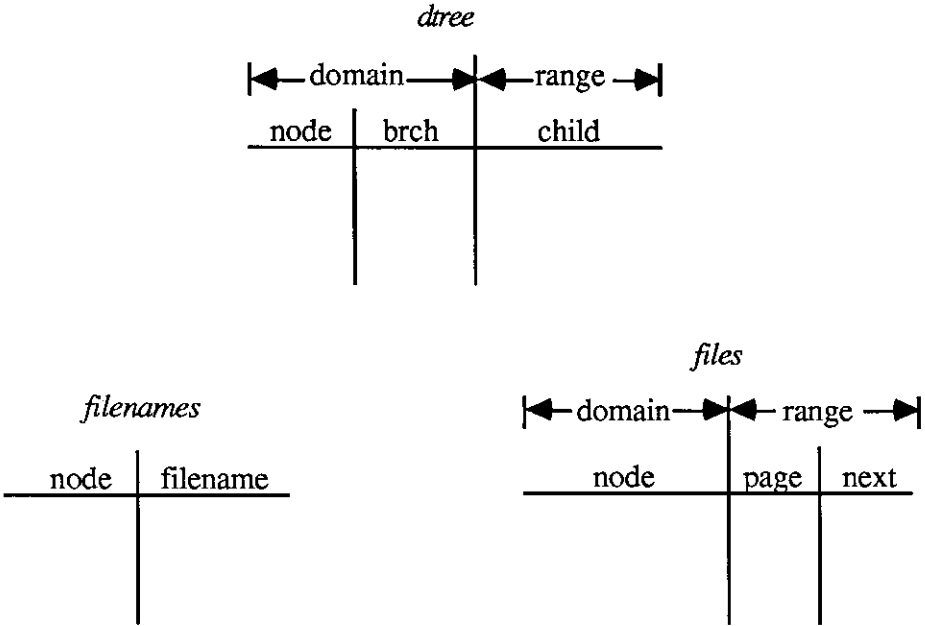


Figure 4.9. Filesystem Relations

The 'open' function converts a filename into the sequential list of pages which constitute that file; the input to the function is a relation which lists the full path to that file as a sequence. Given this, the 'open' function is defined as

$$\text{open } path = \text{image } files.\text{range}.\text{path_nodes } path$$

where

$$\text{path_nodes } path = (\text{path_lng } dtree) \cap \text{node_lvl } path$$

$$\text{node_lvl } path = \text{filenames}^{-1}.\text{all } (\text{index } path)^{-1}$$

Chapter Five

Conclusions

This thesis has two distinct goals. The first goal is to show that a database, by unifying data structures, can serve as the kernel of an operating system. The second goal is to show that this type of structure simplifies the operating system and allows it to be built and modified with a minimum of programmer effort.

Both of these goals are met in Chapters 3 and 4. Chapter 3 clearly shows that a database provides a common storage structure upon which any other data structure can be imposed. Using this new basic structure, Chapter 4 demonstrates how the various operating system concepts can be implemented. Judging from the amount of code required, it can clearly be seen that an operating system implementation using a relational database is much simpler and less time consuming to implement than an equivalent conventional operating system. In fact, the functions presented here required only a few person-weeks to implement, rather than the person-months or person-years common in an operating system such as Unix. Further, as shown in Chapter 4, implementing a different algorithm or a different concept also requires little effort.

5.1. The Real World

While it is possible to build an operating system based on a database, is it practical to build one? Several assumptions have been made in this paper to provide an ideal implementation environment, an environment not easily reproducible in the real world. Each of the three major assumptions made in this paper is presented below along with its effect on a physical implementation.

1. *Language* - the first assumption made is that the described relational programming language exists; in fact, the language is still experimental with only simple implementations existing [MacLennan82]. Even if the language were available, however, it has not proved to be an ideal choice as an implementation language. The language is complex and often awkward to use; for example, the restriction to binary relations often necessitates building complex relational structures. These complex structures, difficult to conceptualize and use, could be eliminated by permitting an arbitrary number of attributes in a relation. Several other examples of this type also exist. In general, simplifying the language would greatly add to its usability.
2. *Architecture* - the second assumption made is that a computer exists whose architecture is designed around relational structures; this computer would (1) be optimized for use with the relational programming language, and (2) represent all devices as relations. While such a device does not exist, nothing prevents it from being designed and built. Until this is done, the proposed operating system could be implemented by building a compiler for the language (or alternate language) and a set of low level driver routines which map the conceptual relational devices to their equivalent physical devices. This second approach has the advantage of fast implementation, but would also suffer from poor performance.

3. *Performance* - the third assumption made is that the implementation machine is sufficiently fast. This assumption was made so that performance would not be considered while the feasibility of the relational operating system was being shown. Performance, however, cannot be ignored when an actual implementation is considered; if the CPU is spending most of its time executing the operating system, it is spending little time accomplishing user tasks. Unfortunately, implementing a relational operating system on a von Neumann computer would result in a very slow system. Large quantities of data need to be manipulated, unsuitable for the von Neumann's word-at-a-time architecture.

Does this mean that the concept of a relational operating system is unusable in the current computing environment? The answer is no, for other types of uses exist. While impractical for use as an operating system in a general computing environment, a relational operating system can serve as an excellent development facility for operating systems research; emphasis is placed on design, not on programming. Concepts can be quickly developed into working programs, interfaced with other pieces of the operating system, and tested in minimal time. When testing is complete, the package would be implemented again using more traditional approaches. The fast development cycle permitted by this approach encourages the development of customized operating systems, systems which would eliminate the unneeded overhead imposed by a larger and more general operating system.

5.2. Suggestions for Further Work

Numerous areas exist for further research. First and foremost, a working model of a relational operating system needs to be developed; only when this is accomplished will the true performance and capabilities of the operating system be known. This also

involves research in the area of implementation languages, as there is no requirement that the choice of language be restricted to the relational programming language described. Several high level data manipulation languages, such as SQL, can also be used if additional support is provided by a conventional programming language. In addition, new types of relational programming languages can be devised.

A second area for additional research involves designing new computer architectures; rather than building software systems around existing computer architectures, we should be designing computer architectures which support higher level software systems. As an example, to support the principles proposed in this paper, a computer should be designed which executes a relational language directly and whose device structure appears as a set of relations. Besides the obvious performance advantage, this type of computer would be very easy to program. Alternately, rather than designing new architectures, what would the performance of this type of system be on an existing large mainframe or supercomputer?

The final proposed area for additional research involves determining what other external views of the operating system are now possible given a database underpinning. For example, are there alternative environments, such as a database environment, a programming environment, etc., that can be provided to the user at the operating systems level? Since a relational database can support multiple views of the same data, can multiple environments be provided concurrently? Are there advantages to this type of approach? Obviously, this is the most open-ended of all of the research areas, for it asks that the traditional views, and possibly the traditional roles, of an operating system be re-examined in light of the new power provided by using an underlying database.

Bibliography

- [Backus78] Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." *Comm. ACM* 21:8, pp. 613- 641, August 1978.
- [Belady66] Belady, L.A. "A Study of Replacement Algorithms for Virtual-Storage Computers." *IBM Systems Journal* 5:2, pp. 78-101, 1966.
- [Brinch73] Brinch Hansen, P. *Operating System Principles*. Englewood Cliffs, N.J.: Prentice-Hall, 1973.
- [Chamberlin76] Chamberlin, D.D. "Relational Data-Base Management Systems." *ACM Computing Surveys* 8:1, pp. 43-66, March 1976.
- [Codd70] Codd, E.F. "A Relational Model of Data for Large Shared Data Banks." *Comm. ACM* 13:6, pp. 377-397, June 1970.
- [Date81] Date, C.J. *An Introduction to Database Systems* (3rd ed.). Reading, Mass.: Addison-Wesley, 1981.
- [Deitel84] Deitel, H.M. *An Introduction to Operating Systems* . Reading, Mass.: Addison-Wesley, 1984.
- [Dijkstra65] Dijkstra, E.W. "Solution of a Problem in Concurrent Programming." *Comm. ACM* 8:9, p. 569, September 1965.
- [Gray78] Gray, J.N. "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course* (R. Bayer, R.M. Graham, and G. Seegmüller, Eds.). Berlin: Springer-Verlag, 1979.
- [Habermann76] Habermann, A.N. *Introduction to Operating System Design* . Chicago: Science Research Associates, 1976.
- [Kowalski78] Kowalski, R. "Logic for Data Description," in *Logic and Data Bases* (H. Gallaire and J. Minker, Eds.). New York: Plenum Press, 1978.
- [Maier83] Maier, D. *The Theory of Relational Databases*. Rockville, Md.: Computer Science Press, 1983.

- [MacLennan81] MacLennan, B.J. *Overview of Relational Programming*. Naval Postgraduate School Computer Science Department Technical Report NPS52-81-017, November 1981.
- [MacLennan82] MacLennan, B.J. *A Relational Program for a Syntax Directed Editor*. Naval Postgraduate School Computer Science Department Technical Report NPS52-82-006, April 1982.
- [MacLennan83] MacLennan, B.J. *Relational Programming*. Naval Postgraduate School Computer Science Department Technical Report NPS52-83-012, September 1983.
- [Martin82] Martin, J. *Application Development Without Programmers*. Englewood Cliffs, N.J.: Prentice- Hall, 1982.
- [Ritchie74] Ritchie, D.M. and K. Thompson. "The UNIX Time Sharing System." *Comm. ACM* 17:7, pp. 365-375, July 1974.
- [Tanenbaum76] Tanenbaum, A.S. *Structured Computer Organization*. Englewood Cliffs, N.J.: Prentice- Hall, 1976.
- [Ullman82] Ullman, J.D. *Principles of Database Systems* (2nd ed.). Rockville, Md.: Computer Science Press, 1982.