# DESIGN AND PERFORMANCE EVALUATION OF THE TEXAS OBJECT BASED SYSTEM

by

**GAD JOSEF DAFNI, B.SC., M.SC.**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

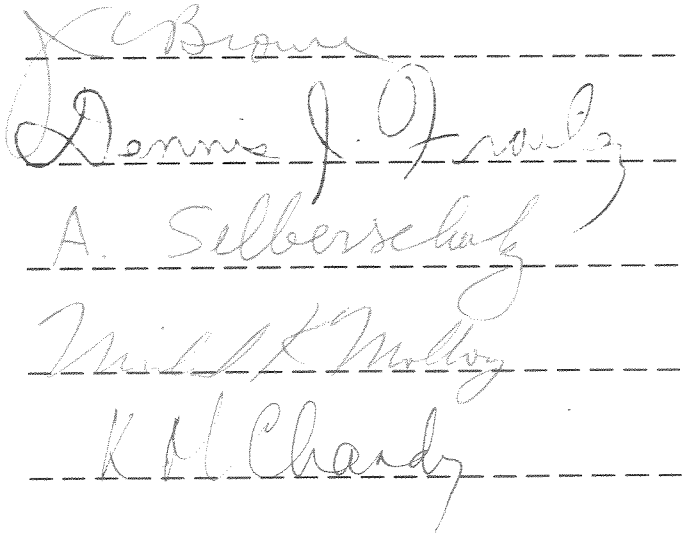
**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1986

# DESIGN AND PERFORMANCE EVALUATION OF THE TEXAS OBJECT BASED SYSTEM

APPROVED BY SUPERVISORY COMMITTEE:



# DESIGN AND PERFORMANCE EVALUATION OF THE TEXAS OBJECT BASED SYSTEM

Gad Joseph Dafni

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-04 January 1987

# ACKNOWLEDGEMENTS

I would like to thank those who helped me to complete this research with their guidance and support. My deepest gratitude is reserved to Dr. James C. Browne for his faithful guidance and inspiration. I am also grateful to Dr. A. Silberschatz for his important help and for providing me financial support through research assistantship. I wish to thank the members of my committee Dr. D. Frailey, Dr. M. Chandy and Dr. M. Molloy for their helpful remarks and advices.

I would also like to express my appreciation to the CS department Unix systems staff that made the completion of this research possible by keeping the Department computers working 24 hours a day 7 days a week.

I thank David K. Bradley for his assistance in programming the assembler and running the early Motorola 68000 tests on the SUN workstation, and P. Horne and Dr. R. Jenevein for running the 68000 tests on the Valid Logic workstation.

Finally, I owe thanks to the National Science Foundation that supported this research by their Grant (MCS-8122039).

The task of formatting this dissertation was greatly eased by use of automated document processing. The Scribe document formatter was conceived of and created by Brian Reid. The current version has been maintained and enhanced by Unilogic, Ltd. The Scribe format definitions for proper dissertation format for The University of Texas at Austin were developed by Richard Cohen.

Gad Josef Dafni

The University of Texas at Austin
December, 1986

# ABSTRACT

Object oriented programming is an approach to reducing software cost by increasing programming productivity, program reliability, integrity and maintainability. This is achieved by bridging the semantic gap between the problem and the hardware concept spaces, that exists in conventional Von Neumann architectures. Achieving this goal involves a significant overhead, which made earlier object based systems inefficient and unable to compete with conventional systems.

This dissertation presents the functional design of an object based machine architecture, which is an efficient and reliable computing element for object oriented programming. Our design avoids the overhead by applying the following principles:

- Early binding of logical entities to their physical properties. This takes place at compile time, link time and at different points at run time.

- Decoupled access/execute processor. The access processor resolves arguments' names, taking advantage of predetermined access modes, while the instructions are executed in parallel by the execute processor.

- Taking advantage of program locality properties. Within the execution locality of a program, the overhead is reduced to a minimum.

Employing these principles involved an analysis of what binding may be done at each step of processing a program, as well as the analysis of the roles of each system component and the exploration of the locality properties of execution domains.

Performance evaluation of the design is done by use of a simulator

that runs on a host machine and emulates the designed system, including its real time. Several benchmark programs were run on this simulator and their runtime was compared to the runtime of the same programs run on other machines. The results show a significant advantage to the proposed design. Combined with the semantic advantages, we believe this system is superior to existing systems.

# TABLE OF CONTENTS

# PART II: EVALUATION

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1

# Introduction

An *Object-Based* System is a computer system where objects are the addressable entities, instead of bits, bytes, words or disc sectors, which are the addressable entities in a conventional computer system. We choose the phrase "object-based" to distinguish this system from *object-oriented* systems, where some hardware support is given to software implemented object accessing. Objects may be defined as instances of scalar or structured types (variables, structures, abstract types), or the abstraction of an instance of a resource. In general, objects are those entities which are of interest to the user, and which are used in the formulation of a problem. These are rarely bytes or sectors, which are typically the entities defined in hardware machines.

There is a vast conceptual span (sometimes called the semantic gap) between the objects definable in current day programming languages and the entities in which these objects are implemented in most current day computer architectures as described above. A large fraction of the instructions executed in current day computer architectures are interpretive realizations of user defined objects. Programming languages are further evolving towards defining objects at a higher levels of abstraction. There are increasing degrees of sharing in concurrent access to user defined objects. The software

implementation of objects and sharing of objects has been largely inefficient and ineffective. There is thus a current and increasing need for computer architectures which efficiently and effectively support direct definition and manipulation of objects. This dissertation reports research on design and evaluation of the Texas Object Based Architecture System (TOBS), which directly, efficiently and effectively implement objects and sharing of objects.

## 1.1 Design Goals

The critical top level design goals are efficient and context controlled resolution of names to structured objects and the values associated with these structures. This high level design goal can be resolved to the following subgoals which are the basis for design of mechanisms:

1. Multilevel software-hardware scheme for binding names, access rights and concurrency control schemes.

2. Object accesses via logical and symbolic names for both primitive and abstract type objects.

3. Flexible but rigorous protection, including the encapsulation of object representation.

4. Provision for concurrency control and sharing.

5. Support for object naming, accessing, and concurrency control in distributed execution environment.

6. Competitive performance for this high level abstract hardware machine.

## 1.2 Conceptual Basis for the Design of TOBS

The design is based upon a coherent synthesis of three principle concepts:

- Incorporation into the total system architecture of multiple levels of binding from names to values.
- Use of a decoupled access/execute architecture.
- Provision of hardware support for exploitation of locality in object accessing.

The subsequent paragraphs sketch each of these design concepts.

Names can be bound to addresses and thus to structures and values at several times in the programming and execution of a computation. Efficiency of access to values usually increases as the binding is brought forward to early phases of the programming or execution process. Binding at the time of actual access in execution involves repetitive interpretation and is thus usually the least efficient but the most flexible of modes of access and control of sharing of data. The essential element of this architecture is identification of the cases where binding can be done early and pushing outward these cases for binding as early as possible. The occasions for binding of structures and values to names include:

- Compilations of the program representing the computation.
- Linking of the compiled program to its execution environment.
- Initialization of the computation.
- At the creation boundaries for different execution domains within the computation.

- As the objects are actually brought from memory to the processor for execution.

These opportunities for binding are displayed in Figure 1-1.

The TOBS architecture has made a series of design decisions which press the binding toward the earlier levels and towards the most efficient possible access at execution times by means of effective exploration of locality. These trade-offs inevitably impact certain other aspects of computational structures. These effects will be discussed fully in later sections.

A computation as it executes, typically passes through phases in which its access is confined to a locality or subset of the total set of objects in the computation domain. This locality property has been explored at the physical level by the virtual memory concept. The smaller the execution domain, the higher the reliability of the computation. The larger the domain, the more readily early binding can be accomplished. The TOBS architecture explores locality in objects accessing by defining domains of moderate dimension, binding as much as possible at the time of domain creation and by creation of efficient mechanisms for addressing within a domain. Accessing within a domain is direct for local objects. Accessing of global objects within a domain is through a nickname table who's entries are resolved as early as possible, either at the domain creation time or at the time of first reference to the object. Thus when possible, repetition of the interpretive binding is avoided. The cost of this efficient binding is the restriction of sharing effectively to procedure or domain boundaries, although "release" mechanisms are also provided (that is, deletion of entries for nickname table).

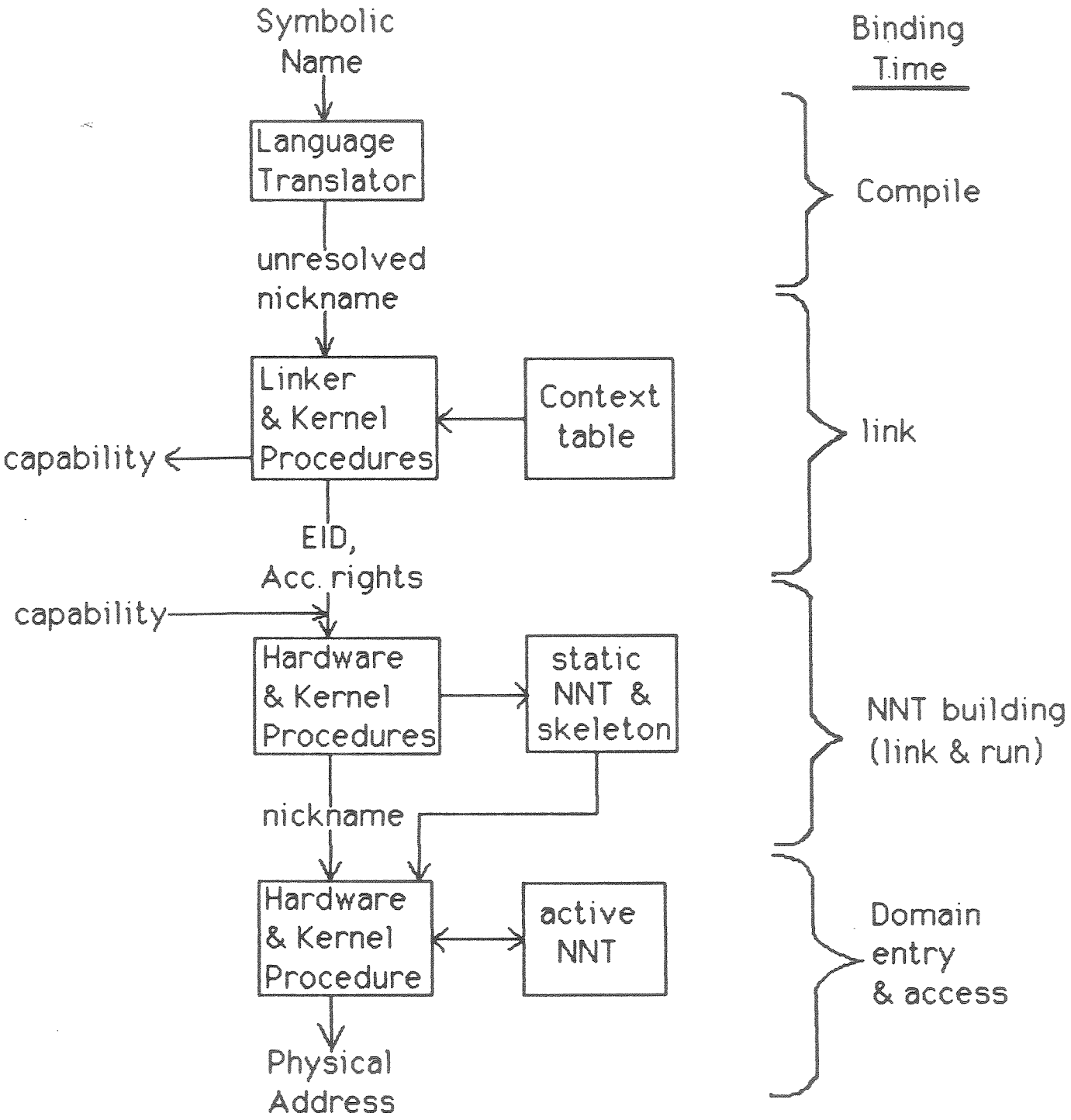



**Figure 1-1:** Name Binding

Most modern architectures do implement some pipeline of instruction execution which implements look ahead of the fetching of operands for instructions. Concepts of decoupling of access processing from execution processing have been developed by Davidson and his students [Pleszkum 83] and by Smith [Smith 82]. This concept has been carried further in this architecture to include a much more extensive set of functionalities, including implementation of sharing and concurrency control mechanisms. This synthesis of concepts has given architecture which is shown by simulation to effectively implement Object Based Access. Each of the concepts is defined and discussed in more detail later in this dissertation.

## 1.3 Review of Past Object Oriented and Object Based Systems

Many of the general ideas mentioned above have been known for more than a decade. They were implemented in several object-oriented systems [Fabry 68], [England 74], [Wulf 81], [Jones 79], [Myers 80], and lately in object-based systems [Houdek 81], [Hemenway 81]. They did not gain much success, mainly because they failed to provide a satisfactory solution to the problem of the large overhead that is involved with trying to close the semantic gap mentioned before. As a result, some of them were very inefficient, while others did not carry the object-access principle all the way, by letting the user direct access to the internal representation of objects, in order to enhance performance (see [Jones 79], [Jones 80]). Yet, some of the basic ideas and techniques that were implemented in the past by software may be implemented in hardware in today's technology.

In this section we will look at some of the main problems, and how they were solved in the above systems.

### 1.3.1 Addressing

The addressing function is the mapping of a logical-name space onto a physical-address space. It is simple when there is a direct mapping between the logical name and the physical address, but this leads to more serious problems (see [Fabry 74]). The simplest case is when the physical address is the object's name, as it was in the **Chicago Magic Number Machine** [Fabry 68]. This makes access to resident and nonresident objects faster than in other schemes, but we pay for it when an object has to be moved, where a search should be conducted for all the places where its address is used. To avoid this, a level of indirection through a System Capability Table (SCT) was used in the **Plessey-250** system [England 74]. The same role is played by GST in Hydra [Wulf 74], [Wulf 81], and by the Global Object Table (GOT, as we shall call it further) in [Lanciaux 78] and [Browne 82].

In order to reference an object, we have to reference its GOT entry. One way to do that is to use the GOT entry's index. This has the disadvantage of a fixed GOT structure, which means that we cannot free GOT entries of objects that no longer exist. We cannot reuse them either, unless we find and destroy all the old references to these entries. That can be a real problem, since as was observed in the Hydra system, 98% of the objects had a very short life span (see [Almes 80] pp. 47), which may result in a GOT fifty times larger than it really has to be. Some other addressing schemes that did not use unique identifiers (UIDs) were investigated by Fabry [Fabry

74] and were found inadequate for addressing shared relocatable objects. Fabry's results are much more significant in distributed systems, therefore the importance of UIDs is much larger in such systems. Of course, the problem of mapping UIDs to GOT entries now arises, but this may be solved by a hardware implemented hash technique.

The number of objects in the system, and hence the GOT size, is usually very large compared with the size of the main memory. This necessitates a memory-management level, which is handled much as in conventional virtual storage systems. There are basically two approaches: *single-level* (flat) and *multi-level* schemes. (We refer here to the mechanism rather than the different levels of accessibility of objects, which in general are not the same).

In the single-level scheme, which was suggested in [Fabry 74] and was used in the IBM System-38, UIDs are mapped directly to physical addresses, by means of a *page-directory* table, which gives the UID and page number (i.e. the virtual page address) of each page resident in main memory. This table is usually found in fast memory and searched either associatively or by a hash procedure. A Non-resident object reference will cause an exception which will bring in the missing object, using the GOT. The single level scheme has the advantage of being simple, straightforward and relatively fast. Yet it has its drawback, which is the size of the address field in the instruction, that must be big enough to contain a UID, and probably a whole capability (that is - access rights too).

To avoid large address fields, another level of indirection is introduced, taking advantage of the locality properties of a program. A Local Name Table (LNT) is used in Hydra [Wulf 74], and *Nickname Table* (NNT) is suggested by Browne and Smith [Browne 82], which contains capabilities (and selectors in Browne & Smith) for all the objects used by a program module. Since the number of objects used by each program module is much smaller than the number of objects in the system, the nickname-table is relatively short and so is the address field of the instruction, which contains an index into the nickname-table (8 bits may be sufficient in many cases).

Some systems, like the ZEUS system [Browne 84], [Browne 85], provide means for dynamic binding of symbolic names. This is added on top of the addressing schemes already described. The symbolic name is translated to a capability by a kernel call. Later the capability or a corresponding nickname-table index may be used. This process is similar to the one used in many file systems, where the symbolic name of the file is translated to a physical address (of a file-table) when the file is opened. This address is usually stored in a variable that is later used to perform file operations on the file. Names are related to a *context-table* (a directory in our example), and may become globally unique if we specify the path to the name (as it is done in Unix file system). A default context-table for each user is also suggested.

The above addressing scheme, including GOT, nickname-table and symbolic-names, will cost one search in a page-directory-table plus two simple indirection levels (name-index and index-capability) per resident object, after

the symbolic name has been translated. There are ways to make this cost as little as possible, and we will suggest such a way later.

There are still several problems related to addressing, besides the main mapping scheme. These are:

- The *small object* problem, that is successfully dealt with by the use of *selectors* in Browne & Smith [Browne 82], which we adopt.
- The generation of names, which is a problem in distributed systems, and is dealt with in [Browne 85] and [Dafni 85].
- The *revocation* problem, which is dealt with in [Redell 74] and by us.
- The garbage-collection problem.

### 1.3.2 Protection

Since objects are the addressable entities, they are also the basic protectable units. It is very natural that the same mechanism that deals with addressing will also deal with protection (as it is with many other addressing mechanisms).

Protection should be rigorous but flexible, so that the principle of minimum privilege may be applied at a low cost. This means that we can change access rights easily, in order to grant each subject the minimum privileges it needs at any time, in order to minimize its ability to do harm.

#### 1.3.2.1. The Access-Matrix

The access-matrix [Graham 72] gives the access rights of all the subjects (users) that may have such rights, to all the objects in the system. Each column contains the access rights of all the users to a single object, and each row contains the access rights of a single user to all the objects. The access rights may include the right to pass certain access rights to some other users.

Manipulating the access-matrix is done by kernel procedures that have the privilege to access it. Each access to an object is verified by the kernel. Knowing the user and the object, the access-matrix is searched, the corresponding access rights entry is fetched and the requested access mode is verified.

The access-matrix method is the most centralized method. It gives the kernel full control of all accesses, thus making it easy to change rights that were granted. The problem with this method is the overhead. First, the access-matrix itself requires a lot of space, since it contains many blank entries (meaning no access rights). Second, the size of the matrix makes the search a time consuming task each time some object is accessed. Third, each manipulation of access rights must be done through the kernel. Fourth, the centralized nature of this method makes it most inadequate for use in distributed systems.

#### 1.3.2.2. Access-Lists

Access-lists are actually the columns of the access-matrix. That is, each object has a list of all the users that have access rights to it. It does not take a special search to find the object's access list, since it may be put in a directory together with the address of the object, that must be found in any case. That makes the search for the access rights simpler than in the case of the access matrix. The access list is smaller than the access-matrix column, because it does not contain empty spaces (for users that have no access to the object). That makes the search even faster.

Access-lists may be distributed according to the distribution of the corresponding objects and their types, which is applicable to distributed systems, and makes the search hierarchical, thus faster.

Maintenance of the access-lists, which means granting and revoking access rights, is done through kernel routines, which is time consuming.

Access lists have an advantage with regard to garbage collection, since it is easy to detect garbage as an object whose access list is empty.

#### 1.3.2.3. Capabilities

The rows of the access-matrix may be viewed as a list of capabilities of a user (subject) to access objects. What is usually meant by *capability* is such an access-matrix entry, which contains the access rights to an object, combined with the object's ID. The user has to present such a capability to a hardware or kernel handler in order to perform an operation on the object.

The handler will translate the ID to the object's physical address and will verify the access rights. In some cases, capabilities will also contain the object's type, which makes it easier to route the handling of the capability to type-managers, where translation, verification and other activity may be done more easily (see [Browne 83]).

To make usage of capabilities flexible, it is desirable to permit moving them around within one's address-space, or transferring them to other processes. This introduces several problems:

- Integrity of capabilities, which means prevention of forgery of capabilities. This must be supported at least by some level of hardware support, that recognizes capabilities as special entities.
- Revocation of capabilities, once granted by a process to another.
- Restriction of access rights of capabilities transferred to another process.
- Amplification of access rights by a process that gets a capability from another one.

The integrity problem has been solved by two methods. One is the protection of certain areas in the virtual address space, that contain capabilities. That means that a user cannot perform unauthorized operations on these areas, and therefore cannot forge capabilities. One can, though, copy capabilities from one such area to another, provided he has access to them. This, together with some other legal operations, provides the flexibility. This method comes in several variants, such as *partitioning*, *fencing* and others. (see [Fabry 68], [Wulf 74], [Hemenway 81], [Jones 79]).

The second method is the protection of certain entities according to their contents, which is called *tagged* architecture. That means that we add some extra bits to each entity, that tell its type, as was suggested in [Feustel 73], [Myers 80], [Gehringer 79] and [Browne 82]. The hardware will not allow performing an operation on the wrong type (as some compilers do). In its primitive form, only two basic types are implemented by hardware - capabilities and data, as was used in the IBM System-38 [Houdek 81]. The tagged method has more flexibility than the protected memory method (see [Fabry 74]), and it allows hardware type-enforcement.

Revocation of access rights is relatively simple when using access-lists. It is harder when dealing with capabilities (see [Fabry 68], [Ekanadham 79]). One common solution is to use indirection, as was suggested in [Fabry 68]. The donor makes a copy of the capability in its address-space, and gives the user a capability to that copy. To revoke the capability, the owner must destroy the copy and never use its address again. Therefore it is suggested that the address of the copy will be a pure logical address, such as a UID. Otherwise we may be losing some more valuable resource. It is desirable that the user of such a capability need not be aware of the fact that it is an indirect capability, which means that it should contain an indirect-reference bit, interpreted by the hardware (or kernel). Another solution may be the use of keys, as was suggested in [Ekanadham 79]. This method has a considerable amount of flexibility, yet it requires variable-length capabilities that can accommodate any number of keys, and more interaction with the kernel, for establishing and revoking locks. Later we will present our own solution.

Amplification of access rights is needed when the principle of encapsulation is applied by a procedure (usually in a type-manager or resource-manager) to its parameter. In that case, the procedure has access to the internal representation of an object, which the caller, that supplied it as a parameter, does not have. The privilege to amplify certain capabilities is restricted to special cases, like those mentioned above. This implies that we do not deal with an hierarchy of authority.

Garbage collection may become a problem when capabilities are used. Reference counts may be used, but they must be protected against crashes (see [Almes 80] pp. 26) and must be accompanied with a garbage collector.

#### 1.3.2.4. Procedure Call/Return

A procedure's execution domain is comprised of three kinds of objects: global, local and parameters. Local objects may be dynamic or static. Static objects will include static variables and constants.

A procedure call is a change of domain. A return from a procedure is change back to the previous domain. These domains may be disjoint or not, depending upon the protection policies and the relations between the procedure and the caller. As we have already mentioned, hierarchy of authorization does not apply in general. The LIFO nature of procedure calls within a single process implies use of some kind of stack. Yet, the conventional linear stack, where activation-records are allocated as stack-frames, is inconsistent with the idea of objects being the basic units of protection, unless the activation records are separate independent objects.

One should also consider the necessity of revoking access rights to such activation-records, once the procedure's execution terminates. It is possible to put a capability for the caller NNT in the NNT of the procedure when it is called, to be used for returning, as suggested in [Browne 82]. Yet it should be a special kind of capability, that may be used only by the *return* instruction. We will not adopt that solution because we like our protection scheme to be connected to the object rather than to specific instructions.

The solution used in the Hydra system seems conceptually appealing. The activation-record is formed as a new LNT (Local Name Table) and its capability is pushed onto the process-stack. That stack is accessible only to the *call* and *return* instructions, therefore protected against unauthorized use. The problem with the Hydra solution is that it is so costly in processing time that each procedure call is considered a major design decision. This is so partly because of the general inefficiency of Hydra, due to lack of hardware support. But the main reason is the overhead of building the LNT each time a procedure is called. In our design we propose a method that uses the same conceptual frame as the Hydra method, but in a more efficient mechanism. Using hardware implementation will reduce the overhead to a minimum.

#### 1.3.2.5. Extended Types

An extended type is an implementation of an abstract type, that extends the set of basic types of the machine. An abstract type is a set of objects, and a set of operations that are allowed on them. It is usually implemented by choosing some convenient representation for the type's objects, and writing a package of functions that implement the operations.

This package is called a *type manager*. Assuming the type manager is safe, then as long as the objects of the type are manipulated only through the type manager (not including copying them), consistency and integrity may be maintained in regard to that type. To ensure this, anyone but the type-manager is denied access to the objects' representation. This property of such packages is called *encapsulation*, which means for data very much the same as *atomicity* means for control.

Object-based systems are most suitable for implementing the encapsulation property, by adding to the list of access rights a right to *seal* or *unseal* an object, which controls the ability to access the internal representation. In [Browne 82] we have instead the right to create component-selectors, which is more powerful than the *unseal* right, since it may be applied hierarchically, to different levels of representations.

### 1.3.3 Concurrency Control

Concurrency control is a way of applying atomicity to transactions on a specific data object. Critical section mutual exclusion is needed because of conflicting accesses to the same data, and the known solutions to this problem are based on controlling concurrent access to certain data objects. Such a control may be applied in some cases prior to execution (see [Taylor 83]), and in other cases it must be applied at run time.

Many systems provide primitives for run time concurrency control. Hardware support for such primitives is essential, since they must be atomic actions. In the following we review briefly some of these primitives, which are

all atomic operations, such that other processes cannot access their operands at the same time. We present our concurrency control primitives in chapter 6.

#### 1.3.3.1. Test and Set

The test and set instruction provides a means for implementing binary semaphores. It tests its Boolean operand. If the operand is zero - it sets it and proceeds. Otherwise - the instruction branches to a specified location. This is a most primitive mechanism that is sufficient for concurrency control. Implementing higher level mechanisms using test and set introduces a lot of overhead. Yet, it is cheap and easy to implement in hardware.

#### 1.3.3.2. Exchange

The exchange instruction is another mean for implementing semaphores. It exchanges the contents of a local variable (or register) with that of a global variable. It has the advantage that the contents may have any value.

#### 1.3.3.3. Decrement and Skip On Zero

This instruction may be used to implement general semaphores, and it is useful for loop control as well. For that reason and because of its simplicity, it is found in many machines. Note that if the counter is required to reside in a processor-local register, the instruction is not useful for implementing semaphores. The fact that the decrement is not conditioned by the operand value causes some problems.

#### 1.3.3.4. Replace-Add

The replace-add mechanism suggested by Gottlieb, Lubachevsky and Rudolph in [Gottlieb1 83] and the fetch-and-add mechanism that was used by Gottlieb et al in the NYU Ultracomputer [Gottlieb 83] has several advantages over the test and set primitive. It is easy to implement in hardware by augmenting the memory controller with an adder, and may be executed concurrently by smart switches in an hierarchical interconnect network. Yet, it has the disadvantage of not testing its operand's value prior to adding, which complicates the implementation of some classic algorithms. On the other hand, the above papers show some interesting algorithms where the power of the replace-add instruction can be exploited for concurrent execution.

#### 1.3.3.5. The HEP System

In the former mechanisms we needed to assign a special object, Boolean or integer, for the special instructions to act upon. Furthermore, the use of these mechanisms was voluntary and anyone could get around them, erroneously or intentionally. In the HEP system [HEP 82] each memory word has its own flag which may be set to FULL or EMPTY. A store operation can be forced to wait until the word is empty before writing it and setting it full, and a read operation can be forced to wait until the word is full before reading it and setting it empty. This mechanism has the nice property that the flag is connected to the data itself, and that its setting, clearing and testing is done automatically and not voluntarily by the user.

# PART I: DESIGN

# Chapter 2

# System's Structure

The system is built of a decoupled access/execute processor, as in [Pleszkum 83] and [Smith 82]. Unlike what is suggested in [Giloi 83], the E-processor is the slave of the A-processor. The A-processor controls all accesses to memory and performs the functions of addressing, protection and concurrency control of objects. The E-processor executes the instructions whose opcodes and operand values are sent to it by the A-processor, and returns the results. Some of the instructions (control and organizational) are executed by the A-processor, sometimes with the help of the E-processor, when arithmetic is needed. Though most computations will be done by the E-processor, the A-processor has the ability to perform address calculations by itself.

## 2.1 System's Block Diagram

Figure 2-1 shows the block diagram of the system.

The main blocks of the A-processor are:

AGV — Address Generator and access Verifier. This unit does all the processing related to object accessing, as will be explained in the next chapters. It will send read-addresses to RAQ, write address to WAQ and will read and write into the addressing tables through the AGV cache.

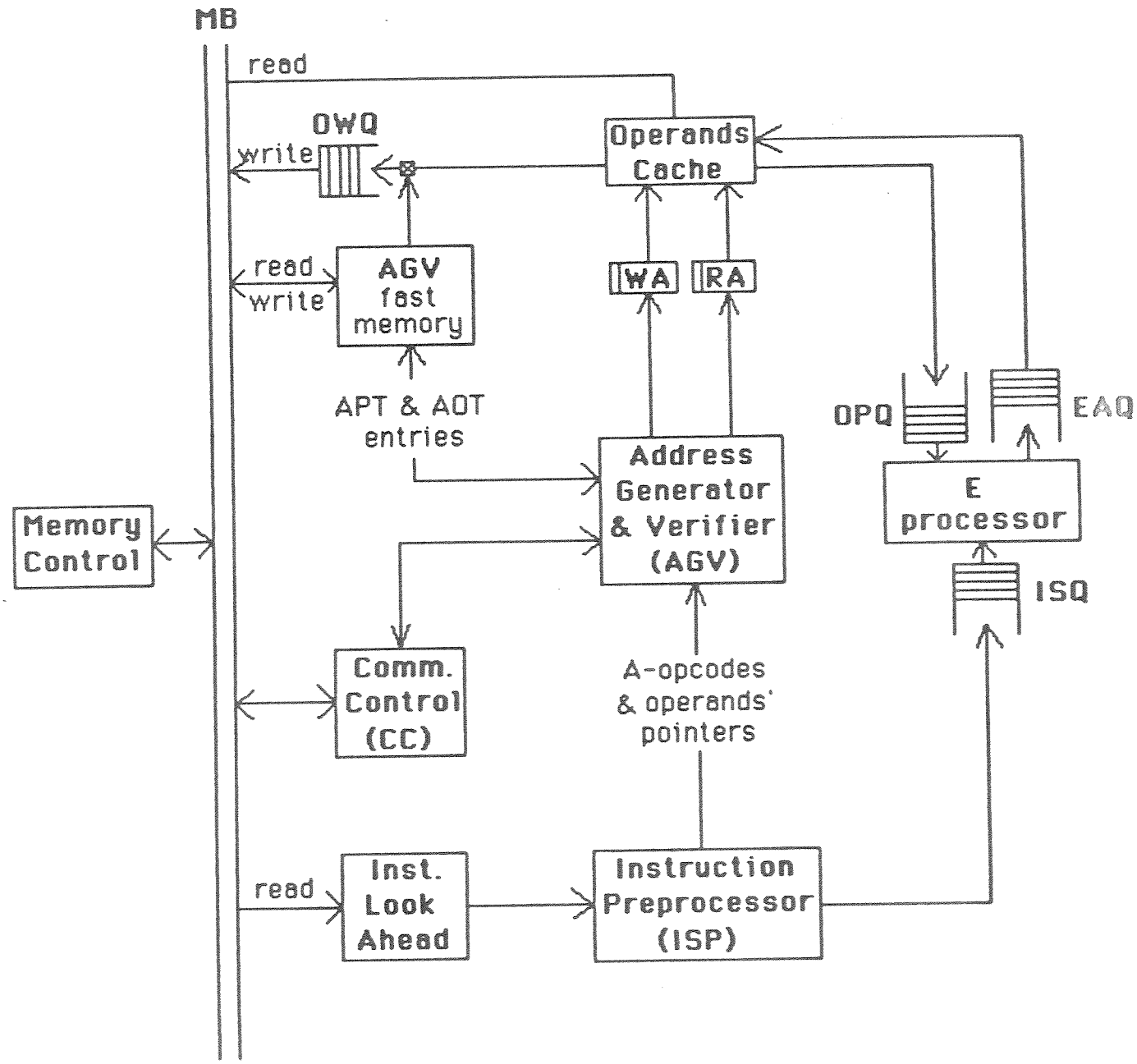



**Figure 2-1:** System's Block Diagram

ISP — Instruction Preprocessor. It will get instructions from RC, separate them into instruction-code and operands selectors and access-methods, send the instruction-code to ISQ and the rest to AGV.

CC — Communication Control. An interface to the communication network. Controlled by AGV through the channel instructions, used by the channel hardware type manager. Has a DMA to memory, that allows data to go only to or from a port object (by reading its tag first).

MB — Memory Bus. Has address lines, data lines and control lines for controlling read/write etc. The memory control acts as an arbitrator to resolve conflicts.

## 2.2 Fast Memories

Fast intermediate memories are used at several points in the access processor to accelerate execution. They are connected to the read and write controllers on one hand, and to the different units of the A-processor on the other hand. Their address spaces are disjoint, so there is no problem of updates from one cache to the other. Follows is their description.

The Address Generator (AGV) Cache

This cache is connected to the AGV and is used for accessing the process stack, APT and AOT for address calculation.

The Operands Cache

Is part of the operands path to/from main memory. Beside the cache it includes an internal control unit that matches a value from EAQ with the address in WA and send them to the cache, or gets a read address from RA, reads the data from the cache and sends it to OPQ. (see description of WA and RA in section 2.4).

The Instruction Look Ahead
: Is part of the instruction path from main memory. Accepts addresses from ISP and sends data (i.e. parts of instructions) back. When necessary, it asks for more data from RC.

## 2.3 Queues

Following are the queues used in the A-processor between its internal blocks and between the A and E processors.

ISQ
: Instructions Queue for the instructions sent to the E-processor.

OPQ
: Operands Queue. Holds the operands values fetched from the operands cache (or memory) before they are processed by the E-processor.

EAQ
: E to A queue. Holds the results received from the E-processor before they are stored in the operands cache and main memory.

EABQ
: E to A boolean queue. Holds the results of boolean calculations in the E-processor, that are used for decision making in the A-processor.

OWQ
: Operands Write Queue. Contains write-through values and their addresses, from the operands cache.

## 2.4 The AGV Registers

The AGV has several internal registers for holding the program status variables. Each register contains three parts. The first part holds the virtual address of some object and the second the third parts hold the relative limits of some active part of the object (like an activation record of a stack). The registers are:

| | |
|---|---|
| LCR | Local code register. Points to the local code segment of the protected procedure. |
| LS | Local stack register. Points to the local stack. |
| DFD | Default data segment. |
| SNR | Static NNT register. Points to the beginning of the static NNT. |
| DNR | Dynamic NNT register. Points to the beginning of the dynamic NNT. |
| LDR | Local data register. Points to the local data segment of the domain. |
| PDR | Process' data register. Points to the process' data segment. |
| DTR | Domain's tag register. Contains an O-pointer to the current domain tag. |
| PS | Active process' stack. |
| PQU | Active process' system communication port. |
| PC | The program counter. Points to the next instruction to be fetched. Size is not applicable. |

Beside these registers, the A-processor has the following internal registers that are not related to the process status:

| | |
|---|---|
| WA | Write-Address register. Holds the next write address generated by the AGV, before it goes to the operands cache. |

WA has a flag that tells whether it has valid contents or not. If not - the AGV cache control will not use it. Before loading WA, AGV looks at RA. If its contents is valid and is the same as the intended WA contents, AGV will wait until RA is emptied.

Note that a queue instead of a register is not beneficial assuming the cache speed is high relative to the AGV and E-processor (that is - assuming negligible cache cycle time). It may only contribute to the queuing space of the corresponding path, and that may be achieved by extending the EAQ. On the other hand, a queue must be searched for conflicts each time a read address is generated, and that is much harder to do compared to a register.

RA — Read-Address register. Holds the next operand's address generated by the AGV, before it goes to the operands cache. Like WA, RA has a valid flag. Before loading RA, AGV looks at WA. If its contents is valid and is the same as the intended RA contents, AGV will wait until WA is emptied.

See note in WA description.

Scratch Pad — Scratch registers for temporary use.

## 2.5 Deadlock Recovery

The communication between the two processors presents the possibility of deadlocks. That may happen when both are waiting on each other while the relevant queues are empty. To detect and recover from such situations, the A-processor will employ a timeout mechanism that is activated after it waits for a time longer than the maximal processing time of any instruction, while the ISQ is not accessed by the E-processor. The timeout procedure will check the queues and send the necessary data to reset the E-processor. Then it will activate a processor error recovery procedure.

# Chapter 3

# Model of Computation

In the following we describe the model of computation for the TOBS system. We start with the general description of objects, then continue with the bigger execution units and the relations between them, and go down to the elementary data units and the instructions. Some of the more primitive concepts are used before they are defined. In such cases we make a reference to the definition. The reader is advised to look ahead when the definition of such concepts is required.

## 3.1 Objects

A TOBS's object is defined as in [Wulf 81], to be an instance of a type. An object may stand for an abstract concept, but it must have a physical representation in memory and a name. Objects are divided into different kinds according to where they physically reside and how their representations are structured.

### 3.1.1 Local Objects

Local objects are objects that are accessible only to a certain protected domain (see Section 3.3) and reside in its local addressing space, or to a certain process. Their names, which are context dependent, are shorter and their addressing is simpler. They are not accessible from outside the domain, which is linked at one time, therefore their access control is managed

by the compiler and linker and not by the run-time system. The only access that needs a special access right in order to be exercised on a local object at run time is the "unseal" right.

Local objects are not internal objects of the domain, but are special external objects (that may have internal objects of their own). The space they occupy is allocated by the system from a local segment, which is an internal object of the domain, or from the the process' segment of the executing process. The local segments include the local stack, the local data segment and the local code segment (where data may be embedded as literals). Local objects are created by the compiler and loader, or dynamically by the use of the CREATE instruction. If they reside in the local data segment, there is no control on which process may use them (which may be the intention). If they reside in the process' data segment, they may be used by any procedure called by that process, but not by other processes that use the same procedure. If they reside on the local stack, they can be used only by the current activation of the domain.

Local objects access control is handled by the compiler and linker. The programmer specifies the access rights to an object together with its declaration and the compiler and linker enforce them. This is possible since a protected domain is always linked to one module before it is run. Concurrency in accessing local objects may occur when different activations of the same domain try to access the same static object. Such concurrency is not handled by the system.

Local objects do not differ in their structure from other objects of the same type.

### 3.1.2 Global Objects

Global objects (sometimes called remote objects) are objects that are accessible to anyone who has their name and the proper access rights for them. They reside in the public domain (that is in no specific protected domain), are named by their UID and accessed from inside a protected domain by their local nicknames (see Section 4.1.3). Note that a code segment can never be a global object by its own, but it is always internal to a protected domain.

### 3.1.3 Segments

A segment is an object which groups together objects that are local to a domain or a process. The segment is protected from unauthorized access, but once a procedure has access to a segment it is allowed to access the objects it contains in any way except for unsealing them. The addressing in such case is relatively simple, and is done by pointers that contain physical displacement.

Segment size may change dynamically in certain cases. In such cases, the segment size that appears in its tag is the maximum size it may have. Entries that appear in the segment's OPT but are not used, are marked as empty. Segment size is limited by the size of the pointers.

Segments may be of one of three types - data segments, code segments and NNT. The segments that are used as part of the execution

environment of a procedure are the local stack segment, the local data segment and the process' data segment which are of the data type, the local code segment which is of the code type, and the static NNT, dynamic NNT and dynamic NNT skeleton, which are of the NNT type.

Data segments are used as pools for allocating space for local objects. This is done both statically and dynamically. The segment contains for that purpose a bit map and a list of partly filled pages.

### 3.1.4 Object Cluster

An object cluster is a page that contains several small global objects, in order to make their paging more efficient. The objects contained in an object cluster may be of different types. Their connection to the cluster is loose and the system may move an element out of the cluster when it is convenient. The difference between a cluster and a segment is the amount of independence they provide to the objects they contain, and their sizes. Unlike in a segment, each object in a cluster has its own GOT entry and its own protection.

The user will not deal with clustering, since the cluster has nothing to do with the logical characteristics of the objects involved. Clustering will be done by the compiler, linker and the run time system, based on locality properties. The problem of when and how objects will be connected to a cluster is outside the scope of this research. We assume though that the compiler and the loader cluster static objects.

### 3.1.5 Object Structure

Objects may be structured or not. A structured object is an object whose internal representation is defined by means of other object types.

A structured object has two parallel structures, the physical structure and the logical structure. By the physical structure we mean the way physical space is allocated for the object. By logical structure we mean the way the object is built from its logical components. Both structures are defined by the object's tag, that includes its type, its length, a pointer to its page table (OPT) and its logical description (OLD) that depends upon its type. Objects that are shorter than a page size will have null OPT pointer. Objects whose type implies that they are shorter than a single page, will not have an OPT pointer at all.

OPT is a system object that contain a list of the page frame addresses where the pages of the object reside. It defines a mapping of the object's page numbers to physical page frame addresses. OPT itself may occupy more than one page, in which case it will have its own OPT. An internal object's OPT may be the same as the OPT of the external object that includes it. An OPT entry includes the permanent page frame number (PPF) of a page in the encompassing segment, or the physical page frame if the object is global. If the page has a copy in main memory, the page pointed by PPF may not be updated. Yet, it may serve as a backup of the page on the auxiliary memory, which is usually more stable.

An object's OLD is an object of OLD type whose entries point to the elements of the object. An OLD entry format is:

PN — Page number in the external object, where the element begins.

D — Displacement in page to the beginning of the element.

Ptag — The offset of the element's tag in the external object tag.

OLD is a one dimensional array of P-pointers. It is either included in the object's tag or has a category 2 O-pointer in there. It provides an efficient mean for finding the offset of the object's elements from its beginning.

### 3.1.6 Segmented Objects

A segmented object is an object that is divided into several independent segments (that have their own names). These segments may be dynamically allocated to the object, and may change size.

The tag of such object contains an OST - *Object Segment Table*, which contains O-pointers of the segments. The object size field in the tag refers to the size of the tag itself, which usually is contained in a single page. If the tag is too big, it is paged and the object has an OPT. The segments have their own OPTs, if necessary.

Beside the OST, the object tag may contain an OLD, but in this case its entries are of the form <S,PN,D>, where S is the segment number. The PVA of the pages is related to the segments to which they belong.

### 3.1.7 Distributed Objects

The elements of a distributed object may reside at different sites. We will describe such an object as a collection of independent objects that have something in common. That something is the possibility of defining them as elements of the same object (besides the possibility of accessing each of them independently, by its own name). We implement a distributed object by giving each of its elements a UID and a GOT entry at the site where it resides, and creating an *object's root* which is of OBD type and contains a table of descriptors for all the elements, which have the same format as GOT entries. This object's root stands for the distributed object as a whole, and resides at one of the sites. The preceding structure may be nested to any depth.

## 3.2 Addresses

Object addresses are expressed in two ways - as pointers by the user, or as O-pointers for the system's use.

### 3.2.1 Pointers

A **pointer** (or a *local pointer*) is an object of special type, that points to another object, or contains the other object. Pointers are used inside a protected domain (though the may point to remote objects). They can also be used as parameters for remote procedure calls. Pointers, unlike O-pointers, are seen by the programmer. The difference between pointers and capabilities is that they are context dependent. That means they depend upon specific NNT or local segments. Therefore they have no meaning outside this context. Some of them that refer to dynamic NNT entries that may have different meanings at different activations, should be reevaluated at each activation.

Besides its tag, a pointer contains an addressing mode M and an address A. The addressing mode M tells the address section where the address A is to be interpreted, and whether it is direct, indirect or immediate.

Pointers, like capabilities, may be manipulated only by the use of certain instructions.

See Figure 3-1 for pointer format.


Where:

8 - is the pointer type code.
C - is the class: short, medium or long.
M - is the addressing mode.
A - is the offset.

**Figure 3-1:** Pointer

### 3.2.1.1. The Addressing Mode

In order to avoid complex addressing and validation when they are not needed, we divide the address space of a protected module into several sections, that are distinguished by an addressing mode that accompanies each operand in an instruction. The addressing mode also determines the pointer size and the use of indirection.

The pointer class is determined by its first one or two bits. 0 means short pointer, 10 means medium size pointer and 11 means long pointer. Besides the pointer length, the interpretation of the rest of the addressing mode bits also depends on the pointer's class.

Following the class bits are 3 bits that specify the addressing mode in the class, as follows:

Short pointer:

0 local code segment.

1 immediate data(#).
Size depends on the data object.

2 local stack segment.

3 indirect through local stack (@).

4 data segment (own, static).

5 indirect through data segment (@).

6 direct nickname (EN of the selector).

7 nickname (static or dynamic according to the

MSB of the A-field) (external)

Medium size pointer:

0 local code segment.

1 immediate.

2 local stack segment.

3 indirect through local stack.

4 data segment.

5 indirect through data segment.

6 direct nickname.

7 nickname.

Long pointer:

0 local code segment.

1 immediate.

2 local stack segment.

3 indirect through local stack.

4 data segment.

5 indirect through data segment.

6,7 null pointer (last bit immaterial).

Notes:

- The short data mode is always related to the default data segment.

In the medium and long data modes, the first two bits of the A-field are used for selecting the data segment, as follows:

| | |
|---|---|
| 0x | - default segment. The second bit x is part of the displacement. |
| 10 | - local data segment. |
| 11 | - process' data segment. |

Indirection is applicable only when the pointed object is itself a pointer.

- Each of the address sections will have an internal register pointing to its beginning. The data segment may be local or process' global (see Section 3.1.3). The contents of these registers is part of the process status.

- The direct nickname mode deals with the EN field of the selector. This is regarded as integer and may be used with all the integer operators.

### 3.2.1.2. The A-field

The A-field of a pointer is interpreted according to the addressing mode. It may be a physical offset in tokens from the beginning of the appropriate segment or NNT, or the object's representation itself in the case of immediate mode.

In general, the actual offset is the value of the A-field, except for the following cases:

- The A-field of a nickname is multiplied by the NNT entry size in tokens, to give the offset in tokens from the NNT beginning.

- For stack and data modes we define the lower part of the segments as a virtual "register" pad, and the actual offset OFFSET in tokens becomes:

$$\text{OFFSET} = \begin{cases} A \text{ Lshift } RS & \text{if } A < 2^{**}RN \\ A \text{ Rclear } RS & \text{if } 2^{**}RN \leq A < 2^{**}(RN+RS) \end{cases}$$

Where

"X Lshift N" shifts X N bits to the left

"X Rclear N" clears X's N right bits

2**RS is the "register" size

2**RN is the number of "registers"

and RN and RS are parameters of the architecture (see chapter 7).

This arrangement allows the effective use of short addresses for objects larger than a token, at the cost of some memory waste in certain cases. Use of a cache make these locations equivalent to general registers, to a certain extent.

### 3.2.2 O-pointers

An **O-pointer** is a system's object which is a pointer to an object. That means that it may be resolved to a certain degree to a physical address. Access to O-pointers is not granted directly to a user. O-pointers are created and used internally by certain instructions. O-pointers belong to one of the following categories:

- <0,EID>: An unresolved pointer.
- <1,PVA,d>: A resolved in-site pointer. PVA is the page virtual address (PVA - see Section 4.2.1). d is displacement in page.
- <2,PVA,d,P>: A resolved in-site pointer. P is a pointer to APT or RPT entry (see Section 4.2.5, 4.2.4). d is displacement in page.
- <3,EID,site>: A resolved out-of-site pointer. Site is host hint found in the local GOT. It may be wrong but in most cases will lead to the right site.

An O-pointer is context independent. Unlike pointer or capability, it is internal to the system and the user cannot copy it or use it in instructions.

### 3.2.3 P-pointers

P-pointers are physical addresses. Like O-pointers they are not directly accessible to users but are used internally by the system. They have two forms:

- <0,PF,d>: Object's physical address.
- <1,PF>: Page physical address.

where PF is a page frame number and d is a displacement in the page.

## 3.3 Protected Domain

A protected domain is an object whose internal objects share the same address space and are protected against external interference. The address space of the domain is defined by its nickname table (NNT) (see Section 4.1.3.1) that contains references to all the objects that are outside the domain and are directly accessible from inside the domain. References to internal objects of the domain from outside the domain must be done by a procedure call using parameters, or through the interprocess communication mechanism (see Section 3.6.7), using a capability with the appropriate access rights. Local objects cannot be directly accessed from outside the domain.

An important property of a protected domain is that it is linked at one time. That means that local objects name resolution, access control and concurrency control may be done at compile and link time, using information supplied by the programmer through language constructs.

A protected domain usually contains several procedures and data

objects that are used to implement certain abstractions. Yet, the programmer may draw the lines of the protected module wherever he likes (as long as it contains whole objects), and for efficiency reasons may choose to combine several such abstraction into one domain.

The NNT has two active parts - static and dynamic. A third part, the dynamic NNT skeleton, is used for building the dynamic NNT at activation time. All the NNT parts are protected external objects (see Section 4.1.3.1).

### 3.3.1 Physical Structure

Protected domain is a segmented object. Therefore its tag contains an OST that contains pointers to the dynamic NNT skeleton, the static NNT, the local data segment and the code segment. Its tag also contains an OLD that contains entries for its internal objects. These internal objects are the procedures that are accessible from outside the domain.

See Figure 3-2.

### 3.3.2 Execution Environment

The address space of a protected domain is divided into six sections:

- A local code section.
- A local stack section.
- Two data sections.
- Two remote addressing sections, static and dynamic.

The local sections of the address space consist of names that are called *local*

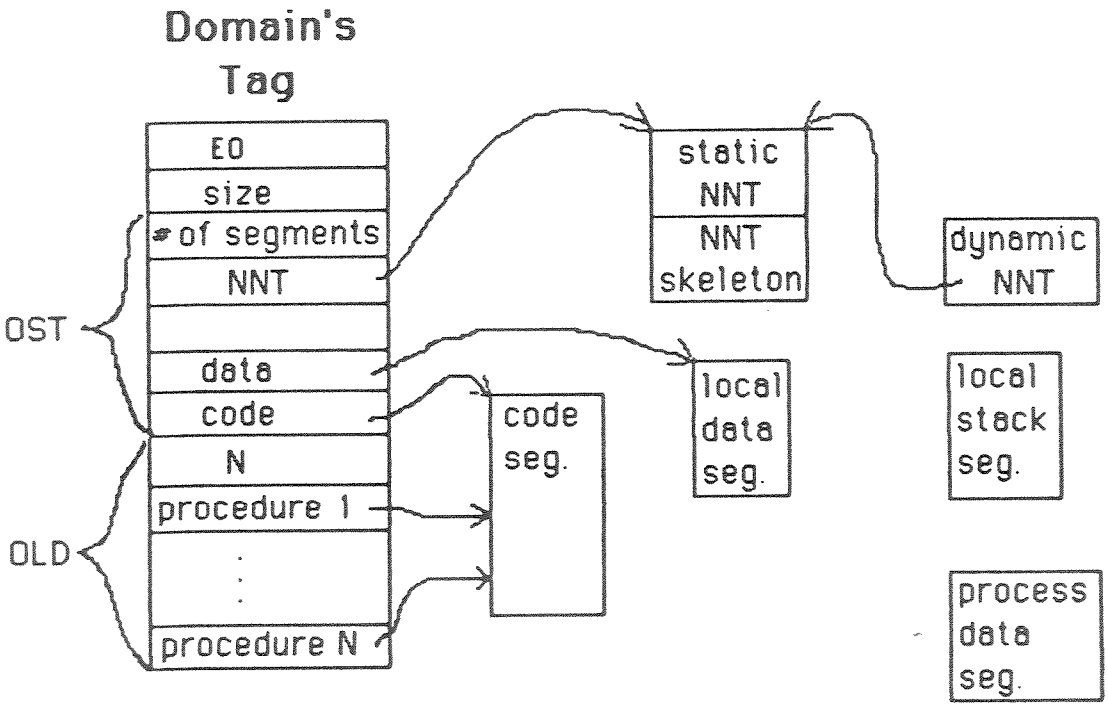



**Figure 3-2:** Protected Domain

*names* and are comprised of a segment identifier and a physical offset. Objects addressed by local names are not protected. The remote addressing sections consist of names that are called *nicknames* and are comprised of the NNT identifier and an index into the static or dynamic NNT.

### 3.3.3 Internal Registers

The access processor has several internal register pairs for holding the program status variables, as described in section 2.4. At context switching, these registers together with some working registers that are used by the microprogram are stored on the process stack and the process stack pointer is stored in a system queue. At domain entry, the above registers are also stored on the process stack, then the same process executes some procedure of the domain, as is explained in the following section. Beside these registers, each of the processors has a set of internal scratch registers.

## 3.4 Procedure

A procedure is an object that is represented by a code segment and a data segment. It is always an internal object of some protected domain. There may be several procedures in a protected domain, some that are accessible from outside the domain and are regarded as its internal objects, and some are local.

### 3.4.1 Call

A procedure call will have two operands, given by pointers. The first operand is a pointer to the called procedure and the second is a pointer to an actual parameter table (ACP).

The ACP contains a list of pairs <ARO,pointer>, where the pointer (see Section 3.2.1) points to the actual parameter, while ARO is an access rights object that contains the access rights that the caller wants to grant the procedure. The ARO of certain entries may be missing, which is interpreted

by the CALL instruction as a request to grant to the procedure the maximum access rights that the caller has to the parameter. If the ACP entries are static, the compiler builds it and take the parameter's access rights into account in the *module's required access rights* (see Section 5.1) of the caller. It also compares the parameter's possessed access rights to the caller's *required access rights* (see Section 5.1), which in the case of a local object are assumed to be unlimited, to make sure that ARO does not include more access rights that the caller has. The compiler or the linker also compare the ARO to the *object's required access rights* of the parameter in the called module and decide whether to abort execution immediately, temporarily disable access rights verification for that parameter or delay action for the actual access time. In many cases the access rights to the parameter can be verified when the caller is entered, but in some cases they need to be verified at the time of the call, or may be even at access time.

If the called procedure is addressed by a local addressing mode, it means a local procedure call. In that case no access rights checking and no address domain change will take place. The procedure will be executed with the NNT of the caller.

If the called procedure is addressed by a nickname, it means a remote procedure call. The NNT entry indexed by the nickname is a selector for a procedure in a protected domain which is pointed to by the selector's father. The selector contains the procedure number as its element number (EN). The procedure entry process is started if the access rights include the *enter* right.

### 3.4.2 Entry

During a procedure entry process initiated by a local procedure call, the PC, stack registers and the local parameters are stored on the active local stack. Then a new activation record is opened on it, the stack registers are adjusted to it, the number of actual parameters is put in its first entry, pointers to the actual parameters (which may be immediate, direct or indirect) are put on the following stack entries, the effective value of the first operand is put into PC and execution proceeds.

During the procedure entry process initiated by a remote procedure call, the internal registers (not including the scratch) are pushed on the process stack, and a new dynamic NNT is built from the dynamic NNT skeleton and the ACP. The dynamic NNT skeleton contains a formal parameter table (templates in [Wulf 74]). The hardware will copy the actual parameters from the ACP to the dynamic NNT while substituting parameter nickname by the actual caller NNT entry. If this entry is not a parameter entry (in the caller's NNT), the calling domain's UID is entered into the NNT entry as the father's UID. If the entry is already a parameter entry, it is copied as is (see sec4.1.3.1). If the parameter is given by a local pointer, it is assigned an NNT entry (in the new domain's NNT) in which the parameter's O-pointer (in terms of the segment's UID and PN) is included, instead of the father's NNT and OBP. Type consistency with the formal parameters types is checked and access rights are verified. The result is a parameter entry being put in the dynamic NNT for each parameter, and its nickname being put on the local stack.

Access rights verification for the parameters is done as follows. The access rights of the actual parameter are merged with those of the corresponding formal parameter, thus may be amplified. Then they are compared to the *required access rights* of that NNT entry, as compiled by the compiler. Entry to the procedure is denied if any parameter has less access rights than the *minimum required access rights*. If a parameter has more access rights then the *maximum required access rights*, the access rights checking for it is disabled for the current activation of NNT. Amplification of access rights may take place only if the procedure has a special capability for the parameter's type manager (see Section 5.4.3).

After building the dynamic NNT, the hardware generates a new active local stack, opens an activation record on it and puts the number of parameters and a local/remote flag in its first entry. Then the internal registers are set to their new values, the effective value of the first operand is put into PC and execution proceeds.

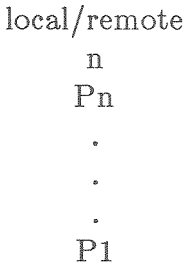



**Figure 3-3:** Local stack after call

### 3.4.3 Parameters Accessing

After a procedure call, the local stack contains the number of local parameters n at stack location 0 (see Figure 3-3). That means that there are n parameter pointers stored in the n stack locations -1 to -n, where parameter Pk's pointer is stored at stack location k-n-1. Therefore parameter Pk is addressed by using stack addressing mode with a negative displacement k-n-1. The hardware checks that $k \leq n$.

If n=0 it means that there are no actual parameters.

Remote calls generate parameter entries in the dynamic NNT, with nicknames in the local stack.

Parameters may be called by value, using the immediate mode in the ACP entries, or called by reference, using the other addressing modes.

### 3.4.4 return

The *return* instruction will have no operands. It destroys the current activation record on the local stack, then restores the program status from the local stack. If the local/remote flag is set to remote, a remote return takes place, the internal registers are restored from the process stack and the local stack object is destroyed. Otherwise, the stack registers are restored from the local stack.

Returned values will be stored in the caller address space through pointers supplied as parameters.

## 3.5 Type Manager

A type manager is an object that implements basic operations on other objects, including the creation and deletion of objects. Type managers for primitive types are implemented in hardware (or firmware), while extended type managers are implemented as a special kind of protected domain. In both cases, the type manager must have several tables that describe the characteristics of its functions to the compiler. These tables are the *access rights table* and the *concurrency keys and masks table.* The tables of the primitive types are part of the compiler itself, while the tables of extended types are defined in the type manager source code and the compiler saves them in a special file.

### 3.5.1 The Access Rights Table

The access rights table for a type is a list of the primitive access rights needed for using each of the type's functions. The entries appear in the order of the type's functions (the entry index is the function's opcode) and contain a bit pattern of the primitive access rights (see Section 5.1).

### 3.5.2 The Concurrency Keys and Masks Table

The concurrency keys and masks table is a list of the keys and masks used with each of the type manager's functions. The keys are bit patterns divided into fields, that are added to the object's lock when it is accessed. The masks are used to test the lock for being locked. They have the same field structure as the keys, and have a 1 in the MSB of some of the fields, depending on the function (for more details, see Section 6.2).

## 3.6 Process

A process is a schedulable unit of computation. It is represented by a pointer to its *process environment table* (PET - see later) that appears in one of the system's queues. Its operations are performed by the modules it activates. Its address space as a process is defined by its capability list, that is a part of its environment, by its data segment and by its stack. The modules that it uses may have access to objects that are not included in the process' capability list.

See Figure 3-4.

### 3.6.1 The Process Environment Table (PET)

The process environment table is a system object that contains four capabilities: to the process' profile table, to the process' stack, to the process' main module and to the process' data segment.

The process' profile table contains a capability to the characteristics table of the principal (see Section 3.8) on whose account the process runs, and a capability to a capability list, whose "copy" access right is disabled. The profile table is a system object that cannot be manipulated directly by the process. The capability list is an array of capabilities, whose "copy" access rights are disabled.

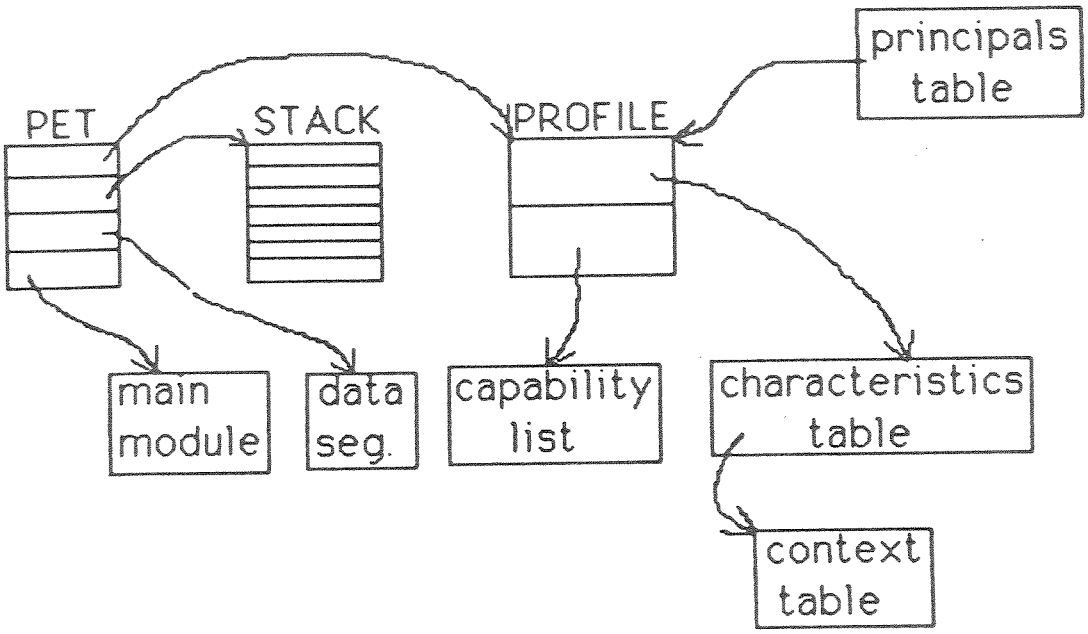



**Figure 3-4:** Process Structure

### 3.6.2 The Process Stack

The process stack is an object that is generated by the operating system when a process is initiated, and is owned by the operating system. It is used for storing the process status at contexts switching and at a protected procedure call. In both cases it is manipulated by the hardware. Since the local stack registers are stored on the process stack at a protected procedure call, it logically connect them to one big stack for the process. Yet, each local stack is regarded a different object with it own protection.

### 3.6.3 The Main Module

The main module of a process is a procedure in a protected domain that is activated by the system at the time the process is created, and stays active (in the sense that it is not being exited) as long as the process exist. The main module's relative address in the domain is put on the process stack where PC is normally stored, at process creation. The main module's activation is granted certain capabilities by the process creator, which it later can grant to other domains. These capabilities are given in the process' capability list, whose capability can be used only to be copied to the NNT.

### 3.6.4 The Process' Data Segment

Contains data that is global to all the domain activations that are activated by the process.

### 3.6.5 Process Creation

Except for the system's bootstrap process, every process comes into existence by another process. When it is created by the use of the CREATE instruction, the "process" type manager executes a procedure call to procedure-1 of the main module of the process (whose UID is part of the initial value for the created process). The new process does not automatically inherit any of its creator's capabilities. Instead it is supplied by its creator with a capability to a PET that contains a capability for a capability list, as part of the initial value of the process. After that the process stack is initiated and cleared. Then the status of the new process it entered into its stack, a pointer to this stack is entered to the PET and a pointer to the PET is entered to the system's active queue.

A process may be created locally or globally. It is created locally if the CREATE instruction is given a local pointer. In such case, the PET is created locally in the segment specified by S and a local pointer to it is returned. Scheduling of local processes is not done by the system's scheduler, but by a local scheduler that divides the processor time allocated to the father process among itself and its sons by any policy it likes. That means that only global processes has entries in the system's queues, while local processes has entries in their father's queues that has to maintain them. In any case, each process has its own process stack and capability list. Notice that a process that created subsidiary processes while executing some procedure in a protected domain, cannot leave this protected domain while any of its subsidiary processes is living. This is enforced by the fact that the RETURN instruction for exiting the domain can be executed only by the father process. If the subsidiary process is dynamic (i.e. its PET is allocated in a dynamic segment) it is automatically destroyed at this point. Static process may be continued when the domain is entered next time.

### 3.6.6 Context Switching

By context switching we mean changing the process on which a processor is currently running. This action is done by the use of the SWITCH or WAIT instruction that is executed from an asynchronous procedure (see Section 3.6.7.1) of the scheduler, or from the running process itself. The result in either case is the storing of the current process status on its stack, putting a pointer to its PET in the active or waiting queue and activating a version of the scheduler on this processor. The scheduler, that is part of the process' father, selects the next process to run on this processor from the waiting queue

that it maintains. By using the SWITCH instruction, it loads the processor status from the process stack, while partly resolving the domain and segment pointers. The system's scheduler's code and tables are shared simultaneously between all processors of a multiprocessor, and should be protected against inconsistency. Local schedulers schedule a time slice that was provided to them by their father on a specific processor, and therefore are not subject to simultaneous execution by several processors.

The above implies that there is no central scheduling and the scheduling process is independent of the number of processors, which may change while the system is running, as long as a processor does not go down in the middle of running a process. Adding processors presents no problem. A newly added processor is exactly in the same state as any other free processor. It will start running the scheduler and select a ready process from its queue. The number of processors need not be known and their action is coordinated by the use of the scheduler queues.

### 3.6.7 Interprocess Communication

Interprocess communication is done by sending messages through unidirectional logical channels. A logical channel is an object that is represented at each of the processes it connects by a communication port, which is a buffer or a queue of buffers, together with some control information that is used by the channel type manager. The channel type manager includes operations for opening a channel, closing it, sending and receiving messages and testing channel's status.

In order to open a logical channel to a process whose nickname is P, process Q uses the instruction "CONNECT C,A,P", where A is the requested properties of the channel, like the queue size and the message receive policy, and C is a pointer to where a capability to the port is to be returned. The type manager, which is implemented in hardware, allocates the necessary buffer and queue and notifies process P through its system channel that it is requested to open a channel with process Q. Process P may refuse (if it has the right to), or else send back its requested channel properties, which may be different from those of Q. The process that initiates the channel creation provides the type manager with a name of an asynchronous procedure (APC - see Section 3.6.7.1) which is later being called automatically when the channel overflows or when it is used without the other side consent. If the process does not provide an APC name, a default system's APC name is used. The process may know about these pathological situations in advance, by reading the channel's status.

A process can send one object per message, yet this object may be pretty large. In such a case, it is divided into packets of the size of a page. In front of the object, the message includes a label of type integer, whose interpretation is left to the channel users. This label can be read by the CHTEST instruction, without reading the message. In many cases the user may find it useful to send a capability for an object rather than the object itself. This may be less useful if the object may be changed before the recipient of the capability used it, or if the recipient resides at a remote site.

The message receiving policy may be by polling (which may be done by a special subordinate process) or by an APC (see Section 3.6.7.1). In polling, the process itself tests the channel's status every once in a while. Using APCs means that the type manager triggers an APC to the process every time a "send" is applied to the channel by the other party. This method gives a better response time, since APCs may occur even when the process is not running. In both cases, a channel handler should exist inside the process, that interprets the received messages and send messages to the other side at certain occasions.

On SEND, a process need not to wait for the message to be received at the other side. It has to wait only when the queue is full. On RECEIVE it has to wait if the queue is empty. To avoid waiting when undesirable a process can use CHTEST before either SEND or RECEIVE.

A communication channel may extend between two processes running on a single multiprogrammed processor, on two different processors in a multiprocessor or in a distributed system. The program does not have to know about that (though timing may differ in different cases). Therefore, the process concept together with the logical communication channels provide program independence of the system configuration.

#### 3.6.7.1. Asynchronous Procedures

Asynchronous procedures are procedures that are part of the process, but are event driven. That means that the hardware issues asynchronous procedure calls (APCs) as a result of certain internal or external events.

Asynchronous procedures run asynchronously and may be in parallel with other parts of the process. Synchronization may be introduced explicitly but is not enforced. There may be different asynchronous procedures that handle different events. The hardware calls the appropriate asynchronous procedure that handles the event that occurred. Priority levels are assigned to events, to resolve conflicts. Asynchronous procedures may be called while the process is inactive. An APC may be called even in the middle of executing a microprocedure, at the points where the microinstruction POLL appears. If needed, the microprogram should save partial results in internal registers that are saved on the process stack, before executing POLL.

Internal or external processes that initiate APCs are connected to logical channels, through which data about the event may be received. The name of the asynchronous procedure and its priority are sent to the channel type manager when the channel is opened, and the type manager will use this name to initiate APCs (see also Section 3.6.7). For each process and priority level there is a queue. These queues are checked by hardware at context switching and at return from an asynchronous procedure. External processes may have direct access to certain queues, and may put there an APC request.

#### 3.6.7.2. System Communication

The operating system is represented by a process that never terminates. Communication between the operating system and another process is done as in the case of any two processes, by logical channels. The system channels are used by the system to inform the process about nonfatal exceptions, like overflow, where the process is to decide what to do about. The

process may refuse communication to the system, which means that it leaves the decision in the hands of the system.

## 3.7 Job

A job is a special process that is created by the system when a user logs on. The job inherits the characteristics and capability list of its principal (see later). The job begins with calling the system's command interpreter which then reads commands either from a given file object or terminal object and interpret them. Each job has its own incarnation of the command interpreter,which represents it. The above may evolve in creating several processes, directly by the command interpreter or by one of the processes created by it.

## 3.8 Principal

A principal is the embodiment of the abstract notion of a user. It is represented by a profile table that contains a capability to the characteristics table of the principal and a capability to a capability list. The characteristics table contains attributes and privileges of the user. Some of the attributes are used as a key for identifying the user. Privileges include the groups to which the principal belongs. The characteristics table also contains billing information. The system will have a principals index table, in which all the principals known to the system will appear, together with pointers to their profile tables.

A principal is not an active entity (unlike jobs and processes). The system may create jobs on his request, one or more at a time. These jobs inherit its characteristics and privileges.

## 3.9 Data Types

Data types are supported by hardware (primitive data types) or software (extended or abstract data types). As in [Browne 82], each object will have a tag in front of it, that contains its type, size and structural data.

Follows are the description of the primitive data types.

### 3.9.1 The Primitive Data Types

| | |
|---|---|
| Integer | 0 size value<br>The value is in two's complements. |
| Real | 1 size sign Esize exp mantissa<br>The exponent is biased by 2**(Esize-1). Mantissa size is size-Esize. |
| Boolean | 2 size value |
| String | 3 size value |
| Template | 4 value |
| Condition | 5 value. A single token object. |
| Pointer | 8 mode value |
| Capability | 9 UID ARF<br>where ARF = DF mask [UID1] is the access rights field and DF is the deferred bit. |
| Access rights | A ARF |

Mark B

Void C size
Specifies an empty block of the given size.

Structured object DX size value
where X is the subtype code extension and the subtypes are:

Array D0 PTP N bound-1 ... bound-N component tag value
where PTP is a pointer to the object's page table, N is the number of dimensions of the array and bound-i is the bound of dimension i.

Record D1 PTP N OLD component-tag1 ... component-tagN value
where OLD is the object's logical descriptor.

Context D2 PTP N value

Pr-table D3 PTP n name profile UID.
Principals table.

NNT D4 PTP size value
Nickname table.

ACP D5 N value
Actual parameters table.

Load module D6 PTP size value

OBD D7 PTP size value
Object directory.

An entry in an OBD type object (an object descriptor) includes:

- A UID.
- A pointer P to an object, that is the object's physical address if L>0 and its host hint if L=0.
- The object's length L.
- The object's total reference count TRC, that is used in garbage collection.

| | | |
|---|---|---|
| | OPT | D8 PTP N value |
| | Data Seg. | D9 PTP size value |
| | Code Seg. | DA PTP size value |
| | Free list | DB |
| | Sizes list | DC |
| System object | EX size value<br>where X is the subtype code extension and the subtypes are: | |
| | Prot. Domain | E0 size OST OLD |
| | Type Manager | E1 size representation-template OST OLD procedures-tags |
| | PET | E2 P-profile P-stack P-main P-data |

| | |
|---|---|
| Profile | E3 value<br>Profile table. |
| Message | E4 size label object. |
| O-pointer | E5 value |
| P-pointer | E6 value |
| UID | E7 value |
| Comm. Port | E8 size table queue<br>where table contains control information and queue is used to hold messages. |
| Queue | E9 size pointer pointers-array |
| Ch-table | EA N context principal<br>Characteristics table. |
| OLD | EB PTP N value |
| Extended type | F type-UID object |

## 3.10 Access Modes

An access mode (access pattern) defines the way elements of a structured object are selected for access during a particular operation on that object. It specifies how the address of the next element to be accessed is calculated. This calculation and bringing the element into the cache memory is done automatically after the last element is used. The access mode is

dependent on the structure of the object and on the performed operation. For the primitive data types, there are some primitive access modes, implemented in the AGV hardware. External access modes may be defined for external types or as extension to the set of primitive access modes for primitive data types.

Access modes are specified in the AM field of the selector for the object's element. It may be given as a primitive access mode code, or as a pointer to an extended access mode. Primitive access modes are coded 0-7. Extended access mode has basic code of 8, plus a (relative) pointer into NNT.

### 3.10.1 Arrays

Beside the trivial access mode that accesses elements in arbitrary order, arrays have the

"From...To...By..."

access mode. When a selector with this access mode is generated, the first and last element numbers and the increment are entered to the selector, the first element's address is calculated and the element is brought to the cache. Later when the selector is used, the current element is used and the address of the next one is calculated and it is brought into the cache.

### 3.10.2 Records

Records will have the field select mode, where a specific field is selected, and operations like "next" and "back".

### 3.10.3 External

External access modes are defined as software procedures that identifies the next element's number to the access processor. These procedures are identified to the compiler and loader as access processor procedures. They are executed by the A-processor in order to fetch the next element. The access processor's hardware will take advantage of special cases, like adjacent element. The loader will translate the external access mode procedure name into a pointer to where it is loaded, wherever it appear. It will also be possible to have an external access modes library.

Example - a binary tree extended type, with the "right" and "left" operators on the selector. Access modes may be "go left" and "go right".

## 3.11 Instructions

The instruction set that is suggested here is not optimized in any sense, since it is not the aim of our research. It was chosen to be simple enough to put no special burden on the design, yet powerful enough to provide reasonable basis for performance evaluation and comparison, without degrading the general performance.

### 3.11.1 Operands

An operand is a pointer without the type token. The instructions preprocessor strips the operand off the instruction and pass it to the address generator. In case of a null pointer, the AGV repeats the last address it generated.

In the following section, whenever an object is mentioned as the operand of an instruction, a pointer to it is meant.

### 3.11.2 The Instruction Execution

Instructions reside in main memory and are fetched and executed one by one by an interpreter written in microcode and residing in the A-processor's control store. The interpreter has two register files. One is the internal registers file and the other is a scratch registers file. It also has access to a simple ALU used for address calculations, a sequencer for timing signals and inputs and outputs to different queues and fast memories. These queues and fast memories has their independent control.

The instruction execution cycle is as follows:

1. Fetch opcode pointed to by PC into a register.

2. Fetch first operand and calculate its address, by calling a microcode procedure. This procedure steps PC past the operand.

3. Fetch the first operand's type code and call the microcode procedure that implements its type manager.

4. If the type manager is primitive, and if the access rights for the object fits, it branches to the proper microcode procedure according to the opcode.

5. If the type is extended, the type manager of extended types is called. It will fetch the extended type manager's UID from the first operand, translate it to a physical address, copy PC to ACP pointer (which is a register used by the CALL instruction) and activate a CALL to the type manager's procedure whose number is the opcode. Then it copies the ACP pointer, that was stepped past the instruction operands, back into PC.

### 3.11.3 The Instruction Set

The instruction opcodes have different meanings for different types of operands. Many of the instructions described in this section have the same opcode as other instructions that act on a different type of objects. We preferred to give them different mnemonics when the meaning was different. Abstract types has their own meanings, which is defined in the type manager. The type manager includes in its tag a list of the procedures that implements its operations, and the opcode is used by the hardware as an index to the table in the tag of the type manager of the first operand, for calling the appropriate procedure. The instruction's operand list is used as an ACP with maximum access rights (i.e. the *required access rights* of the caller) for each operand. If a type manager has more functions than the maximum that is allowed, an explicit procedure call must be used for the extra operations.

The operands are always evaluated by the A-processor, while the instruction is executed in most cases by the E-processor. We will point out when it is not so.

We divide the instructions into groups, according to their function.

#### 3.11.3.1. E-Processor Instructions

The instructions in this section are executed by the E-processor.

In the following, x=TOS and y=TOS-1 of the E-processor's stack.

PUSH [z] Pushes z to the E-processor's stack. If there are no operands then TOS is duplicated.

| | |
|---|---|
| POP [z] | Pops TOS into z. If there are no operands then TOS is not saved. |
| EXCH | Exchange x and y. |
| PUT z | Put z in TOS in place of what was there before. |
| GET z | Get TOS into z, but leave the stack unchanged. |
| CLRS | Clear stack. |
| ADD [z] | If no operands then x+y → y and x is poped, else x+z → x. |
| SUB [z] | If no operands then y-x → y and x is poped, else x-z → x. |
| CMP [z] | If no operands then 1,0 -1 → y according to whether x>y, x=y or x<y, and x and y are poped, else compare z vs x. |
| MUL [z] | If no operands then x*y → y and x is poped, else x*z → x. |
| DIV [z] | If no operands then y/x → y and rem(y,x) → x, else x/z → x and rem(x,z) → x. |
| CLR | 0 is pushed onto the stack. |
| NEG | x is negated. |
| INC | x+1 → x. |
| DEC | x-1 → x. |
| AND [z] | If no operands then y and x → y and x is poped, else x and z → x. |

| | |
|---|---|
| OR [z] | If no operands then y or x → y and x is poped, else x or z → x. |
| XOR [z] | If no operands then y xor x → y and x is poped, else x xor z → x. |
| NOT | not x → x. |
| TRUE | true → x. |
| FALSE | false → x. |
| SHIFT [n] | x shift n → x. If no operand then y shift x → y and x is poped. |
| ERESET | Reset the E-processor. |
| OPQCLR | Clear OPQ. |
| EAQCLR | Clear EAQ.a |
| CLRS | Clear E-processor's stack. |

#### 3.11.3.2. A-Processor Instructions

The instructions in this section are executed by the A-processor.

| | |
|---|---|
| MOV x,y | Move contents of object x to y and make x NULL. |
| COPY x,y | Make a copy of object x in y. |
| AINC x | Increment integer x. |

ADEC x — Decrement integer x.

AADD x,y — Add integer x to integer y.

ASUB x,y — Subtract integer x from integer y.

ACMP x,y — Compare integers x and y. Performs x-y without storing the result.

ASHIFT n,x — Shift x n bits left. Negative n means right shift.

ACLR x — Clear x.

CHDSEG n — Choose data segment according to n, as follows:

| | |
|---|---|
| 0X | - no change. |
| 10 | - local data segment. |
| 11 | - process' data segment. |

ENTNNT P,NN[,CP]

Generate a static or dynamic NNT entry for an object whose pointer is P, at the entry whose pointer is NN. The compiler evaluates the value of NN at compile time and creates a dynamic NNT skeleton entry for it. NN is used later as the object's nickname.

If the dynamic NNT skeleton entry is for an external pointer, an external pointer is created. It is local or remote depending on whether P points to a local object or to a capability for a remote object. In case of capability, its access rights should not be less than the minimum required access rights for that object. If they are more than the

maximum required access rights, or if P does not point to a capability, the access rights checking for that object is disabled (see Section 5.1). After the NNT entry is generated, the capability pointed by P is destroyed.

If the dynamic NNT entry is for a selector, a selector with P as its element number pointer is generated. If the father is not of primitive structured type, we need the operand CP, which is a pointer to a capability for the type manager of the father, and should include the "unseal" right.

AM is set to direct access, with EN=0.

INITST O,SZ[,V] Initiate dynamic object O in the local stack to size SZ and optional value V.

NEWPET OP,CS,NP

Create a new PET NP and a profile table from an old PET OP and a capability sublist CS. An entry in CS contains an index into the old capability list and an access rights mask. The capabilities in the new profile table's capability list are those selected by CS, with their access rights masked by the corresponding access right masks.

This instruction does not need the "copy" access right of the capabilities in the capability list.

This instruction is executed by the A-processor.

INDAR AR,NC,OC

Create an access rights object AR and a new indirect capability NC pointing to AR, from the capability OC.

This instruction is executed by the A-processor.

### 3.11.3.3. Control

The instructions in this section are executed by the A-processor. The BE CN instruction needs the assistance of the E-processor for evaluating its condition.

BA CN,A — Branch to local address A if condition CN is satisfied by the last result in the A-processor. CN is a pointer to a 4-bit condition object whose bits are interpreted as follows:

| | |
|---|---|
| 1 | result was 0. |
| 2 | result was negative. |
| 4 | result was positive. |
| 8 | result set the carry. |

Combinations are treated as the OR of the corresponding conditions.

BE CN,A — Branch to local address A if condition CN is satisfied by the TOS of the E-processor. CN is the same as in BA.

DBNZ x,A — If x>0 then decrement x and branch to A.

CALL name,acp — Call a procedure (local or remote) pointed by "name", with actual parameter table pointed by acp.

ENTRY n — Saves n tokens on the local stack for dynamic variables. These are not parameters, which are stored at negative offset by the CALL instruction, but variables that are declared in the procedure and their sizes are known to the compiler when the procedure is compiled.

RETURN — Return from a procedure.

WAIT CN,x,y — This actually is a context switching to the system's scheduler process. The current process becomes ready again when the condition CN is satisfied by x-y while the scheduler is active. The condition CN is the same as in BA, while x and y are pointers to objects whose values are evaluated each time the scheduler become active. They may include clock reading or acceptance of messages from other processes.

EXIT — This is actually a WAITNE 0,0 instruction. The scheduler recognizes it and kills the process.

ACTIVATE P — Puts the process whose nickname is P into the active queue. The NNT entry whose index is P must include the "activate" access right.

CREATE O,T[,LC]

Create an object specified by the template T. The created object has an undefined initial value.

If O is a pointer to an (empty) capability, a global object is created in the public domain and a capability for the object is returned in in the empty capability. The capability contains most of the access rights, not including the right to create selectors for its elements (the "unseal" right).

If O is a pointer to a local pointer, a local object is created in the local stack segment or the local data segment according to O. A local pointer to it is returned in the pointer pointed by O. There are no explicit access rights involved, but access to local objects is not restricted, accept for the possibility to create selectors for their elements (unseal them).

LC is a capability with the "locality" access right. If it is given, the GOT entry for the new object is opened in the

same locality as that of the object pointed to by LC. If LC is not given, the locality ID of the new object is determined automatically by the hardware (see Section 4.2.2.1).

DESTROY O — Destroy an object. O is the object's nickname. The user must have a "destroy" right for the object.

EXECUTE P — Execute a single instruction at a location pointed by the pointer P, in the current context.

### 3.11.3.4. Type Conversion

CONVERT x,y — Copy object x to object y while doing type conversion.

SYMTCAP S,C,P,PF

Convert a symbol given by the string S, to a capability C with the appropriate access rights (see Section 4.3.2.5) and a path P, which is an array of element numbers. PF is a capability for the profile table that is used, together with the object's entry in the context table, to find the access rights that may be granted. The profile table contains some subset of the identifiers of the groups that the principal belongs to.

This instruction is executed by the A-processor.

UIDTCAP U,C,PF

Concatenate access rights to the UID U, and returns it as a capability C. The access rights are found as above.

This instruction is executed by the A-processor.

### 3.11.3.5. Communication

CONNECT C,A,P

Opens a communication channel with port C and attributes A to process P. C is a pointer to where the capability for the port should be returned. If it is an output channel, the nickname P must point to an NNT entry that include the "write" access right. The attributes A tells whether it is input or output, if it is to call an asynchronous procedure, its name, etc.

SEND M,C — Send a message M to a port whose nickname is C. The NNT entry should have the "write" access right.

RECEIVE C,M — Receive a message M from port C.

CHTEST C,S — Reads port C status into S. The status include the number of messages in the queue and the label of the first message.

CCLEAR C — Clear channel C. Kill all waiting messages.

### 3.11.3.6. System Instructions

These instructions deal with system's objects and perform operations on entities like processes, pages and O-pointers, in order to implement system's policies. In the case of context switching, it may be done at different levels on top of the system's scheduling, so that a process may schedule its subprocesses within the time slot allocated to it.

The instructions of this group are executed by the A-processor.

SWITCH P,Q — Switch context to process P in queue Q. P and Q are defined by their nicknames. The instruction stores the active process' statue on its process stack and loads a new status from the process stack of the next process on the Q queue.

RSWITCH P,Q Return from procedure, then switch context to process P. Useful in asynchronous procedures.

RESPNT P,O Takes the unresolved O-pointer pointed by P and resolves it to the resolved O-pointer pointed by O.

BRINGP P,PF Makes resident the page whose page frame is pointed by P. That may involve the writing of some resident page to auxiliary store, according to the page replacement algorithm. The RPT entry for the evicted page is moved to APT. The page frame in main memory is returned in PF.

### 3.11.3.7. Selector Operators

The instructions in this paragraph are executed by the A-processor alone.

SET S,n,p1,p2 Set selector's S EN field to n and AM parameters to p1 and p2. If it causes a selector type change - all the sons of the selector are disabled (if there are any).

SRESET S,n,CP Resets a (disabled) selector S' EN field to n and enables the selector.

n is a pointer to the element (new) number and CP is a capability for the type manager of the father, and should include the "unseal" right. The father's pointer is taken from the old selector. If S is enabled, CP is not checked an the effect is as in SET.

STEP S,n Steps selector S n elements forward or backward, depending on n's sign. If it causes a selector type change - all the sons of the selector are disabled (if there are any).

NEXT S If the next element of the selector S according to its access mode exist, steps S to it.

NXB S,A — If the next element of the selector S according to its access mode exist, steps S to it and branch to A.

REP S,A — Same as NXB but without stepping S.

### 3.11.3.8. Input Output

No input output instructions are provided at the macro instruction level. For most objects, the user will not have control or knowledge where they reside. Other objects that stand for physical entities, like terminals, will be defined as system objects of special types, and the user will access them like they are part of the objects space.

At the microcode level there are I/O microcode procedures that are being called by the paging instructions.

# Chapter 4

# Naming and Addressing

Object naming and addressing are the main problems in an object based system. By object naming we mean the way we single out one specific object, in such a way that it will not be mixed up with other objects, and can be easily accessed with the appropriate access and concurrency control. By addressing we mean the actual procedure of accessing the object. That usually involve the binding of physical properties to logical names, as implied by the names themselves. This procedure should be efficient without hurting the logical properties of names or their flexible binding.

## 4.1 Naming

In order to make addressing efficient, we deal with two kinds of names - local and global. Local names are used inside a single program module for objects that have no access from outside that module. Therefore they are assumed not to need protection and concurrency control. The programmer may define the modules as he/she wishes. Global names have the form of symbolic names, capabilities and nicknames. The first two carry the following properties of the specific object:

- A systemwide unique identifier of the object.
- Access rights to the object.

Nicknames identify objects and their properties in a certain locality. The different forms of names depend also on the degree of resolution they underwent.

See Figure 1-1 for name binding.

### 4.1.1 Symbolic Names

Symbolic names are those that are usually used by the programmer. The symbolic unique identifier of an object may refer to different context tables in some hierarchy, like the use of directories in some file systems. There is also a default context for each user.

The symbolic access rights are the names of the access rights that the user wants to include in the name, out of those he is entitled to (like read, write, execute etc.).

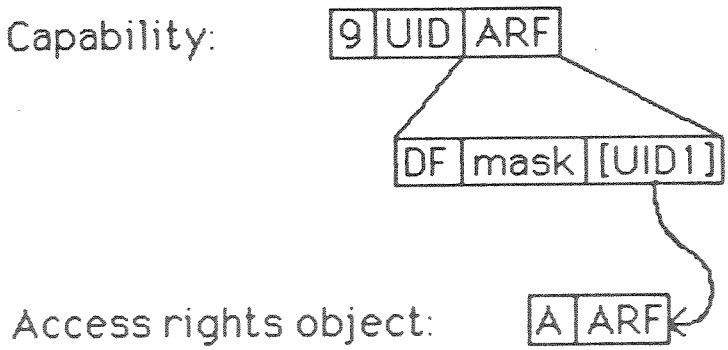
### 4.1.2 Capabilities

Capabilities are a compact intermediate form of names, between symbolic names and nicknames. They are independent of any context or physical binding and system configuration, in the most general sense. That make them transferable between processes that may reside at different sites without need of transformation.

As in the symbolic name, capabilities contain the object's identifier and access rights.

The object's identifier is in this case unique over the whole system (all the sites) and over the lifetime of the system (UID).

The access rights field of a capability (ARF) may be deferred or not. If it is deferred, it contains a pointer (UID) to another ARF, and an access rights mask, which is a bit pattern that specifies what access rights are disabled (bit=0) by this level of indirection. If not deferred, it contains only a mask.




Where:

| | |
|---|---|
| 9 | is the capability type code. |
| A | is the access rights object type code. |
| mask | is an access rights bit pattern. |
| DF | is the deferred bit. |

**Figure 4-1:** Capability

### 4.1.3 Pointers and Nicknames

Pointers are local names to either local or global objects. They are interpreted in a specific context, provided by the execution environment of a protected domain. They are short compared to capabilities, and in most cases are simpler to resolve.

Nicknames are pointers to global objects. They take advantage of the locality property of a program module, in making global object names short. That is achieved by the use of the nickname table (NNT), which is used for translating nicknames to UIDs. Nicknames does not carry access rights, therefore are even shorter.

#### 4.1.3.1. The Nickname Table (NNT)

The nickname table (NNT) is the table of special type that defines the local address space of a protected domain. It contains (indirect) pointers to all the objects that are accessible by the module. The pointers are called *external pointers*, *selectors* and *parameter selectors*, and are entered to the static NNT by the compiler or to the dynamic NNT as parameters entries or by the use of the instruction **ENTNNT**.

NNT entry has a code that tells what kind of entry it is. Some selectors may become temporarily disabled, as a result of changing a selector in the level above them (see Section 4.3.2.3 and the SET instruction in Section 3.11.3).

A disabled selector has code=0.

An **external pointer** is an (indirect) pointer to an external object that may be local or remote. An external pointer has the following fields:

| | |
|---|---|
| 1 | Code for external pointer. |
| LCL | Local bit. Set in a local object's entry. |
| TP | The object's type. |
| SONS | An NNT index of the first son in the list of sons. |
| RAR | Resolved access rights (see Section 5.3.1.1). |
| UID | Unique identifier of the object. In case of a local object this is the segment's UID. |
| AOP | AOT pointer (see Section 4.2.3). |
| PN | Segment's page number if the object is local, or cluster number if the object is clustered. |
| D | Displacement in PN when applicable. |

A *selector* is an (indirect) pointer to an internal object. We call the object it is being part of, its father. The father may be external or internal object.

A selector has the following fields:

| | |
|---|---|
| 2 | Code for a selector entry. |

| | |
|---|---|
| LCL | Local bit. Set in a local object's entry. |
| TP | The object's type. |
| SONS | An NNT index of the first son in the list of sons of the object. |
| AM | The access mode. |
| F | A pointer to its father in NNT. |
| EF | A pointer to its external father in NNT. |
| EN | The element number which the selector selects. EN may be indirect, to accommodate with dynamic element number. |
| SIZE | The element's size. |
| NXBR | An NNT index of the next brother in the list of sons that the object belongs to. |
| OBP | The object's pointer. The page number PN of the external father, plus a displacement D in that page, where the element starts. |
| Ptag | Element's tag offset in the external object. |
| P1 | First AM's parameter. |
| P2 | Second AM's parameter. |
| P3 | Third AM's parameter. |

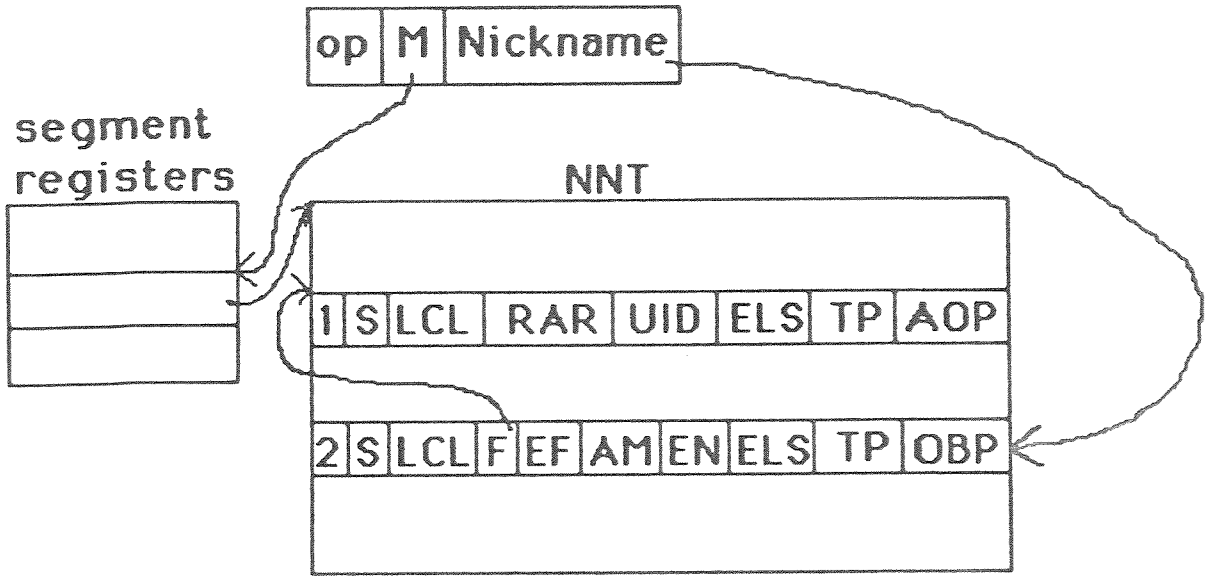
A *parameter selector* is a selector that selects an actual parameter. The father of a parameter selector may reside in another NNT and its resolved access rights may be more restricted than those of its father. If the parameter was given by a nickname, the UID of the domain where the father, as well as the resolved access rights, is included in the selector. If the parameter was given by a local pointer, the parameter selector includes an O-pointer to the local object, instead of the father domain's UID, and the selector's L bit is set. Note that a local object may have an NNT entry in its home domain (specially if it is an internal object), and is given by its nickname, so the first case applies.

See Figure 4-2 for addressing by nickname.

#### 4.1.3.2. NNT Components

NNT has two active components, the static NNT and the dynamic NNT. Nicknames are distinguished by the addressing mode, to point either into the static or the dynamic NNT. The static NNT contains entries whose logical contents does not change at runtime, and has a single copy that is used for all the activations of the protected module. The dynamic NNT entries may point to different objects for different activations, therefore it has different copy for each activation (see Section 3.4.1). The stable part of the dynamic NNT is given in the *dynamic NNT skeleton*. The dynamic NNT skeleton include an entry for each entry of the dynamic NNT.

NNT is built in two phases. The first phase is at link time, when the static NNT and the dynamic NNT skeleton are built by the linker, under




Where:

M - addressing mode.
NNT - nickname table.
S - sons bit.
LCL - local bit.
F - selector's father.
EF - external father.
RAR - resolved access rights.
TP - object's type.
AM - access mode.
EN - element number.
ELS - number of elements
OBP - object pointer.
UID - unique identifier.
AOP - AOT pointer.

**Figure 4-2:** Addressing by Nickname

directions from the compiler. These two parts contain all the entries of the NNT, including entries for parameters, at different stages of resolution. Those parts that can be resolved at compile and link time without restricting flexibility and hurting the logical context are resolved. Other parts, like object's UID or element number in a structure are left to be resolved at later time. The second phase is the NNT activation, when the protected domain is entered at execution time. At this phase a copy of the dynamic NNT skeleton is produced and unresolved items are resolved as much as possible. Parameters are entered into their place, resolved or not. A pointer to the old NNT (which is the contents of the SNR and DNR registers, see Section 3.3.3) is entered to the process stack (which is not accessible to the user, except for the *return* instruction). The dynamic NNT is actually the activation record.

## 4.2 Definitions and Structures

Before we describe the addressing mechanism, we have to make some definitions and describe some tables that are involved in this mechanism. Some of these tables describe the internal structure of an object and are an integral part of it. Therefore the object structure is described as well.

### 4.2.1 Definitions

1. A **reference** to an object is a capability or an NNT (see Section 4.1.3.1) entry. A **passive** reference to an object is an existing capability to that object. An **active** reference to an object is an existing entry at an active NNT, that points to that object.

2. The **extended identifier (EID)** of an external or internal object is the pair <UID,path>, where UID is the UID of the external object that contains our object, and **path** is a list of indexes in that object.

3. The **page virtual address (PVA)** of a page is the pair <UID,PN>, where UID is the external object to which the page belongs, and PN is the page number in that object (that is - the displacement of the page, modulo the page size). Assuming fixed page size, PVA is independent of the site were it resides.

### 4.2.2 The Global Object Table

The global object table (GOT) is an object of OBD type that contains a directory of all objects that have something to do with the local site at the current time. For the structure of GOT entries, look at the OBD type in section 3.9.1.

GOT can be accessed only by kernel procedures that have a constant capability for it.

The GOT of a certain site will include at certain times entries for the following objects:

1. Objects that reside in this site at that time.

2. Objects that reside on other sites but have references from inside this site.

3. Objects that have migrated from this site but still have references directed to its old local address.

#### 4.2.2.1. Locality

In order to decrease the number of page faults caused by accessing the GOT we want it to have reasonable locality. On the other hand we want to use a hash function for accessing it, so the initial access to a locality will not involve too many page accesses. Although these two concepts seem

contradictory, we may enjoy both by dividing the access function into two parts. The first will find the locality to which the entry belongs, using a hash function, and the second will find the entry inside the locality (not necessarily by hash function).

To make this feasible the UID must be broken into two distinct parts - the locality ID (LID) and the object's ID (OID). Both will be assigned to the object at the time it is created, but the user will have the option to request a specific existing locality or a new locality which is assigned by the system. In order to ask for an existing locality, the user has to present a capability for some object in that locality, with the "locality" access right.

The user may prefer to use a default locality. In such case the LID will be built by the system from the user identifier and the sequential part of the object's UID (which is assumed to be in correlation with some global time when the object was created).

#### 4.2.2.2. Extendibility

Since GOT size may change significantly, we will use a dynamic hashing structure as suggested in [Fagin 79]. It has the advantage of very few page accesses in the process of getting to the right bucket. The idea is to use a hash function that generates a pseudokey which is sufficiently big for the maximal number of buckets that may ever be needed, but use only the least significant part of its bits, according to actual need. These bits point to a partition table that contains pointers to the actual buckets. An overflowing bucket will be split in two, which may necessitate the use of another bit from

the pseudokey. In such case the partition table size should be doubled, yet the actual GOT table need not be reorganized (except for the bucket that was split). Using the least significant bits as a partition table pointer makes the doubling of its size easy.

### 4.2.3 The Active Object Table (AOT)

The AOT contains entries for the nonresident active external objects. It is accessed by hashing the UID into a fixed length hash table, that contains pointers into a resident AOT that contains entries. A chaining collision resolution method is used. When the resident AOT becomes full, entries are created in the external AOT, which is a fixed size table in the auxiliary memory. The external AOT is accessed by direct hash, which determines both the page number and the entry in the page. The external AOT is paged by the general paging system.

An AOT entry includes the following fields:

NEXT - index of the next entry in the chain.

BACK - index of the preceding entry in the chain.

UID - Unique identifier.

PF - Page frame number of the object.

D - Displacement in page.

ARC - Active reference count.

LOCK - The object's lock, whose format is taken from the type manager tables while creating an NNT entry.

An entry is opened in AOT when a reference to the corresponding object is entered to some NNT and the object does not yet have an AOT entry. The active reference count (ARC) field is initially set to 1. ARC is used for deciding whether the AOT entry is needed any more. ARC is incremented each time a new active reference to the object is generated, and is decremented when such reference is destroyed. The AOT entry will be destroyed some time after ARC becomes zero.

### 4.2.4 The Resident Page Table (RPT)

The RPT is a list that shows the main memory occupancy at a certain time. The n-th RPT entry tells which page resides at the n-th page frame of the main memory. It contains the following fields:

PI Paging information.

PVA Page virtual address, which is the concatenation of the object's UID and the page number in the object.

PPF Permanent page frame.

RPT is accessed by hashing on PVA into a table that contains pointers to RPT entries, like in IBM-System 38.

### 4.2.5 The Active Page Table (APT)

The active page table (APT) is used to simplify the translation of PVA into page frame, that otherwise may involve an extra page fault while trying to access the OPT. It is a list of nonresident pages that were active lately, or there is a high probability that the are going to be active soon, together with their paging information. An APT entry contains the page virtual address (PVA), its permanent page frame address (PPF) and paging information (PI). APT has a fixed size, which is a system parameter that may be changed at system startup. When a page is moved from memory or when the page look ahead mechanism assigns it a high probability for being referenced in the near future, an entry for it is opened in APT. A hash function on PVA is used for accessing APT. Collisions are handled by overwriting the old entry's contents.

### 4.2.6 The Resolved Addresses Table (RAT)

This is a relatively small associative memory that is used as a cache for RPT. RAT contains the most recently resolved page addresses. Its entry contains PVA and the page frame (PF) in main memory where the page resides.

## 4.3 Addressing

Addressing is the actual action of using the object's name for accessing it. An object may be addressed in different address spaces (or sections of an address space) and at different levels of name resolution. In the following we deal with address space sections by addressing modes, and with address resolution.

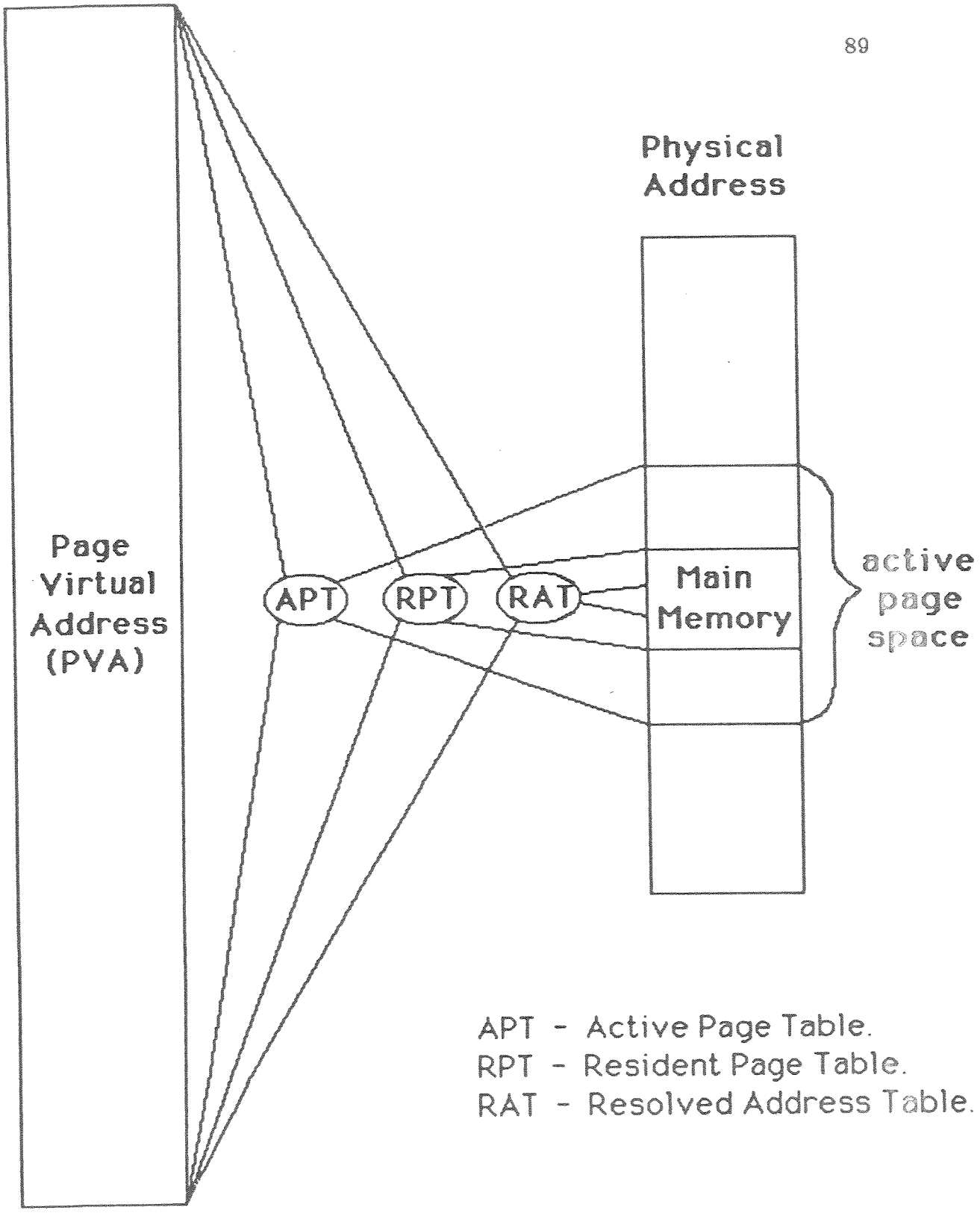



**Figure 4-3:** Page Address Mapping

### 4.3.1 Addressing Modes

Addressing modes accompany each operand and specify the address space section where the operand resides (see Section 3.2.1.1).

### 4.3.2 Address Resolution

Local addressing may be done by direct addressing or by creating an NNT entry for the local object and using its nickname. This is mostly useful for structured local objects. In any case, no hardware checking of access rights or concurrency control is applied to local objects, whose access control is handled by the compiler and linker.

Remote address resolution is much more complicated, since remote objects may be relocated independently of the accessing module and may be shared. This must be taken into account in the addressing mechanism. Remote addressing is done in several levels, each of which takes place at different phase of name to address resolution. We will try to make the most frequently used addressing scheme to be the most explicit one, thus making the average access more efficient. That means early binding. On the other hand, we do not like the binding to be too early, since we may then loose flexibility (in relocating objects) and it may hurt our conceptual basis (by reducing the logical level of symbols). In the following we will start from the bottom, that is from the most primitive way of referencing an object, and go to the top level. At each level we will describe the binding mechanism and the resolution mechanism to the next lower level.

#### 4.3.2.1. Token's Physical Address

At the very bottom level there is the physical address which uniquely identifies the basic addressable physical entity in the system, which is the 4-bit token. The token's physical address is manipulated only by the hardware and firmware, and is not seen even by the system.

#### 4.3.2.2. Object's Physical Address

Objects are built of consecutive tokens, and the object's physical address is the address of its first token, that contains its type code. Object addresses are contained in protected objects, like capabilities and pointers, that cannot be tempered with. Capabilities are generated by the type manager when it creates an object. Pointers are generated by the language translator when it is used for addressing a remote or local object. In both cases the generated addresses can be guaranteed to address the beginning of an object. Since load module is also a protected object that can be manipulated only by the language translator and the loader, we conclude that programs can address only objects.

The problem of unauthorized accessing object representation or addressing into the middle of an object by use of I/O does not exist in TOBS, since I/O can be performed only by the paging system.

#### 4.3.2.3. Pointer Resolution

At the instruction level objects are addressed by pointers, which may be local or remote.

Local pointers are indexes into one of the local segments, that include

the code segment, the local data segment, the local stack segment and the process' data segment. Remote pointers are called *nicknames*, which are indexes into the *nickname table* (NNT). Pointer resolution starts with accessing the data directly pointed by the pointer. That is done by resolving the pointer into a physical address X, using the contents of the appropriate segment (or NNT) register. That segment register contains the following fields:

UID — The segment's UID.

Address — The segment's permanent address.

Lock — See chapter 6.

Size — The segment's size.

The A-field of the pointer is the offset in the segment (maybe after certain manipulation as described in section 3.2.1.2). The high significant bits of it provide the page number PN in the segment, while the low order bits provide the displacement D. The PVA of the page, which is <UID,PN>, is resolved into PF by the general mechanism of resolving PVAs (see Sections 4.2.4, 4.2.5 and 4.2.6), with the help of the segment's permanent address, while making the page resident. <PF,D> gives the address X.

In the case of local pointer, the physical address X is the address of the object (unless the pointer is indirect, in which case another phase of the above algorithm takes place).

In the case of a remote pointer, the address X is the offset in either

the static or dynamic NNT, which together define the remote address space of a protected domain. The following is a description of how the NNT entry is resolved.

We refer to the selector fields as described in Section 4.1.3.1. EN is resolved into an OBP as early as possible. If EN is constant, then it is resolved at domain entry time, otherwise it is resolved at access time. The page number PN and the displacement D of the element are found in the element's father's OPT, using EN as an index. PN is then entered into the selector. When the object is accessed, the PVA (which is the concatenation of the UID from the external father with PN), is looked up in RAT. If not found in RAT, the PVA is hashed on RPT. If the right RPT entry is found, its index is the page frame number PF where the page resides. PF is concatenated with D to form the memory address where the object begins. PF together with PVA is entered into RAT using an LRU replacement policy.

If RAT and RPT entries for the page does not exist, PVA and paging information PI are put in a new entry in RPT that is evicted using some algorithm based on LRU and the expected probability of reference in the near future, and a page fault occurs. A page fault software procedure is activated. This procedure finds the page address either in APT or in the object's OPT using PN as an index, brings the page into memory and opens RPT and RAT entries for it. If OPT has its own OPT (i.e. it is too big), the page fault procedure calls itself recursively, using

PN1=PN div pagesize

as a parameter.

m=PN mod pagesize

is the entry index in the page frame returned, where the object's PF and D are found. The D field of OLD is then copied to the D field of the selector, if not already there. The object size found in its tag is put in the size field of the selector, if not already there. Parts of this process of updating tables and maybe bringing the page, may be done ahead of time when the NNT entry is created, according to the system's policy, based on PI.

The active reference count ARC of the object will be incremented, or set to 1 if it is the first active reference.

When an access mode operator needs to modify its operand selector, it does not have to go through the whole resolution process again. If the next element is in the same page, it has to modify only D. Otherwise it has to find the object in the higher level's OLD, using the OBP pointer found in the higher level's selector. At the same time it will check the selector to be in bounds, using the object's length found in the higher level's selector. If the new element has a different type that the old one, it will also check the S bit of the selector. if it is set, meaning the selector has sons, it will search for them and disable them. This is to make sure that the user got the right to unseal the new element.

Use of cache memory at the proper points may reduce access times (see Section 2.2).

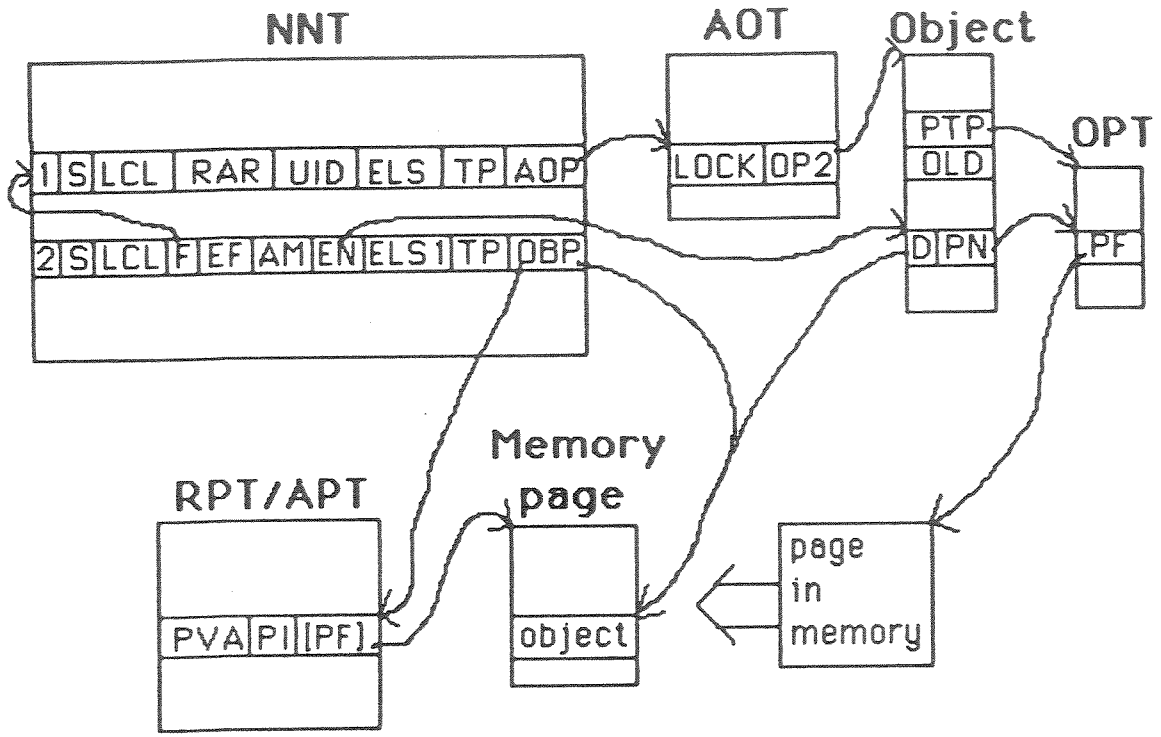
Information for the use of the paging system will be derived from three sources. The first is the time each active page is referenced, the second is the access mode to a structured object and the third is the the NNT contents. The last two may give a prediction on the pages going to be used in the near future.

It should be noted that the external object and page where the object starts may be relocated without having to change NNT entries. The only changes should be made in the AOT and APT, RPT or RAT entries, each at a single place that is known to the system.

#### 4.3.2.4. Capability Resolution

Capabilities are resolved to nicknames, usually at link time but sometimes at run time. Resolving a capability means creating a new entry in the static or dynamic NNT. For the dynamic NNT, this is done by using the instruction **ENTNNT** for the capability and for each element of the path (which may be empty). For the static or skeleton NNT, the loader uses the ENTSNNT, the same way. A capability for the type manager of each of the elements (which is known from the type of its father) with the "unseal" right, is required in order to run the generated load module. These capabilities are entered to the load module's *module's required capabilities* list (see Section 5.1). The path is a list of element numbers, one for each level. The element numbers may be constants or variable (indirect).

The values entered into the NNT by the instruction ENTNNT while resolving an EID are as follows:

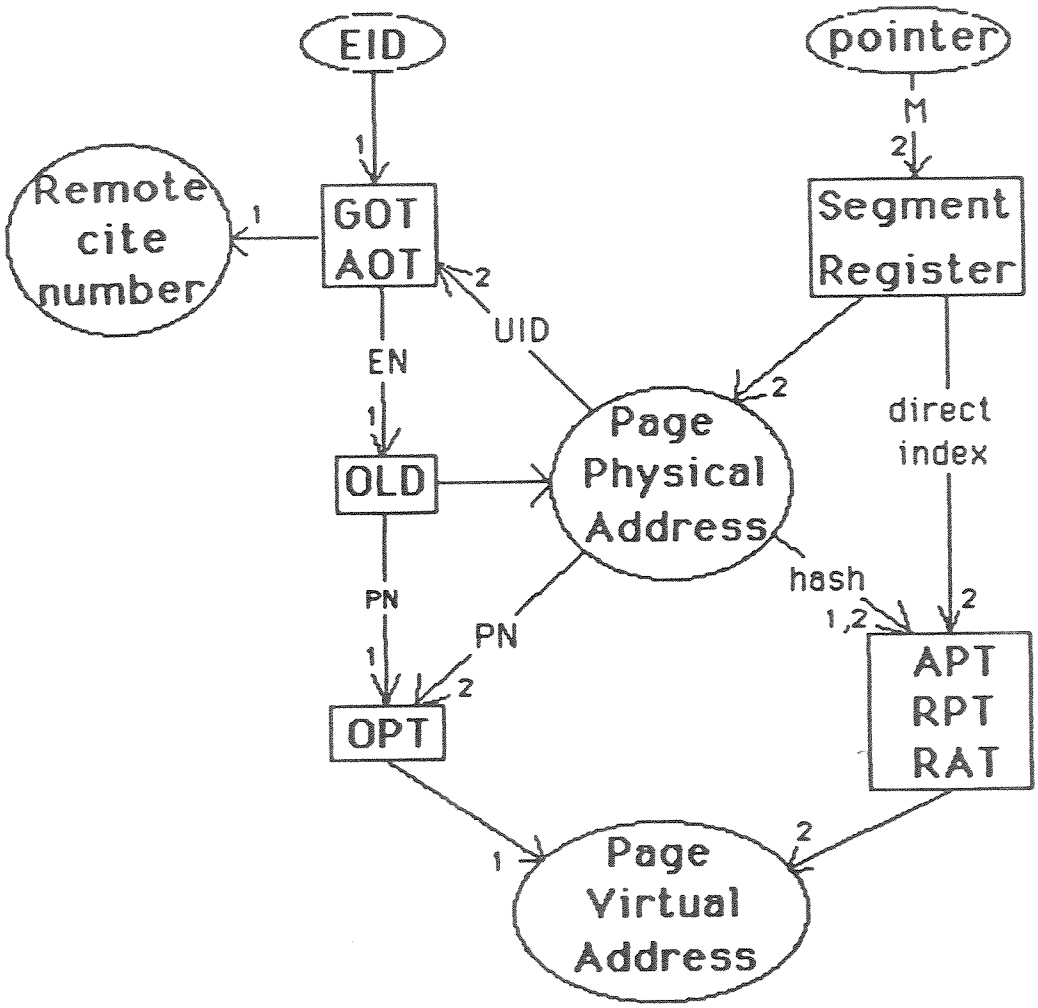



Where:

NNT - nickname table.
APT - active page table.
OLD - object's logical description.
S - sons bit.
LCL - local bit.
RAR - resolved AR.
UID - unique identifier.
ELS - number of elements.
AOP - AOT pointer.
F - father pointer.
EF - external father.
AM - access mode.
TP - type.

AOT - active object table.
RPT - resident page table.
OPT - object's page table.
EN - element number.
OBP - object pointer.
LOCK - concurrency lock.
PTP - page table pointer
PN - page number in object.
D - displacement in page.
PF - page frame.
PVA - page virtual address.
PI - paging information

**Figure 4-4:** Run Time Object Nickname Resolution




GOT - Global Object Table.
AOT - Active Object Table.
OLD - Object's Logical Descriptor.
OPT - Object's Page Table.
PN - Page Number.

APT - Active Page Table.
RPT - Resident Page Table.
RAT - Resolved Address Table.
EN - Element Number.

**Figure 4-5:** Page Address Resolution

- An **external object pointer (EOP)**, that points to the external object that encompasses the object.
- A **selector** to internal objects for each level of subdivision.

The **EOP** is computed from the capability part of the EID. The UID is hashed onto the AOT. If the appropriate entry is not found, the UID is hashed onto the whole GOT and the object's entry is made resident (using the former AOT hash result). A concurrency lock is received from the object's type manager and is put in the new AOT entry, and its ARC field is initiated. Then the object pointer field of it is used to open an APT entry (if needed), and a pointer to the AOT entry is put in the EOP's AOP field.

For access rights resolution, see Section 5.3.1.1.

The selector fields are described in Section 4.1.3.1. The element number (EN) field is taken from the EID's path to the internal object. If the element number is variable, EN will be a pointer to this variable, which may be local or global. In the latter case the pointer is a nickname.

Note that if the NNT is included in the compiled module (i.e. it is the MAIN module), the compiler can resolve nicknames independently. Yet the linker still need to resolve certain items in NNT at link time, though it may not need the returned nickname.

#### 4.3.2.5. Symbolic Name Resolution

Symbolic names are resolved in cooperation between the compiler and the linker. The compiler generates linker directives that include user provided symbolic named or compiler generated symbols, as parameters. The compiler generated symbols include nicknames that cannot be resolved by the compiler.

The instruction SYMTCAP is used to resolve user defined symbols. It resolves the symbolic identifier to a UID by looking it up in a *context table.* The context table name is part of the given identifier, or else it is found in the user's profile table. The context table is an array whose entries has the following fields:

Symbol — The object's symbolic name.

UID — The object's UID.

GLST — A list of user groups with their special access rights. An entry in this list has the following fields:

- GID — A group ID to which special access rights are granted.
- GAR — The group access rights.

WAR — The general public access rights.

The symbolic access rights are translated to a bit mask that together with the permanent access right and the user's group access rights is used to form the ARF field of the capability (see Section 5.3.1.1).

The paths to internal object if any, are resolved as a list of constants, local variables or nicknames.

SYMTCAP returns a capability and a path, which are used by the linker to create an NNT entry, or to be stored directly in the code, in the case where the *capability* operator was used by the user (i.e. x:=capability(name) ). The capability and the nickname are entered into the linker's tables under the symbolic name and the compiler generated symbol for the nickname, to be used later.

### 4.3.3 Remote Site Accessing

When a capability to an object that does not have a GOT entry appears in a site, as a result of either the creation of a new object or receiving of a capability from another site, a GOT entry is opened for the object. If the capability came from another site, a host hint is sent with it. This host hint is entered to the new GOT entry, when the capability is moved from the input message by the capability type manager. The IS flag in that entry is cleared (see OBD type, Section 3.9.1).

When the IS field of the object pointer in GOT is not set, a remote site access takes place. We distinguish between two cases:

1. The destination operand is in-site but some of the source operands are not. In that case a message containing a capability for the object and a "copy" operation code is sent to the site where the object is supposed to reside. The object locator of the remote site will try to locate the object in its address space. If it succeeded, it will call the object's local type manager that will check access rights and concurrency key, and send a copy back. If the object is

not found, the message is passed to where the remote site assume the object is. This process must terminate with finding the object or finding that it was destroyed, since each pointer points to where the object was the last time it was accessed from that site. The object locator of the originating site will pass the object's copy to the type manager, that will complete the instruction execution (for good or for bad). Then the object locator destroys the object's copy.

2. The destination operand is out-of-site, or the operation is a call to an out-of-site procedure. In these cases the operands are resolved locally as much as possible, then the whole instruction, including operation and operands, is sent to the site where the destination operand resides, where it is carried out by the appropriate type manager. After the execution is completed, an acknowledge message that carries the conditions under which it was terminated, is sent back to the originating site, which will take action when required.

The preceding procedure for remote access does not directly involve relocating objects. The amount of remote traffic to an object, and its sources, will be the basis for implementing an object relocation policy.

## 4.4 Virtual Memory

By virtual memory we mean a scheme by which the code and data needed for the computation as a whole are divided between different levels of the physical memory, in such a way that the system will have optimal cost/performance under certain constraints. The physical properties of the memory levels to be considered are its access and cycle times, transfer rate, accessibility (i.e. is it directly accessible) and availability. Availability is dictated by cost on one hand and by the performance and the constraints on the other hand. Usually cost is directly related to speed (access time and

transfer rate). That leads to a hierarchy of memories, where at the top level there is a small fast and directly accessible memory, while at the bottom there is a big slow (and usually more stable) memory. Once memory is available, it should be utilized optimally under the constraints, no matter what its cost. In order to increase performance and decrease delay times, we want to keep as much data as possible in the higher levels of the hierarchy. Since these levels have relatively small capacity, we may be able to keep there only part of the data needed for the computation at a certain time interval. This results in cases where the data we need is not available at the right place, and we need to bring it in (page faults). We like to minimize the page fault rate at each level.

In order to simplify things and avoid fragmentation and compaction, we use a fixed page size for the data moved between the different memory levels.

### 4.4.1 The paging System

The whole physical address space of the system (of a single site) will be divided into fixed size (typically 512 bytes) page frames. The address of the page frame is the physical address where it starts modulo the page size. The address may contain device code and an internal address.

Each object that is larger than a single page is divided into (logical) pages. The logical name of such a page, or its *page virtual address* (PVA) is the pair <UID,n>, where UID is the UID of the object and n is the page number in the object.

A page resides in some page frame in auxiliary memory. It may also have at certain times another version (usually more updated) in main memory. We call the auxiliary memory page frame - the permanent page frame (PPF) and the main memory page frame - the resident page frame (RPF) (see also Section 3.1.5). The RPF, if it exist for some page, is the valid version of the page.

While a page may be allocated different RPFs at different times, its PPF does not normally change. If that is necessary, its OPT entry and maybe AOT entry should be changed too.

An object that includes more than a single page, begins at the beginning of its first page.

An external object whose length is no greater than one pagesize will always be contained in a single page.

Non structured objects (internal or external) will always be contained in a single page. A *void* object will occupy the end of a page if the page is not full (*void* is a special primitive type).

The addressing mechanism evokes the paging system in a look ahead fashion, based on the access mode to objects. As a last resort, demand paging is used. The procedure call instruction also evokes the paging system in a look ahead paging fashion, that may be based on past experience with the

procedure and on its NNT contents. All this information, together with the time since last time the page was referenced, is used for calculating PI of the page, which is its priority for residing in memory.

The paging system, which is basically implemented in software, is supported by the system instructions RESPNT and BRINGP (see Section 3.11.3.6).

Out of site objects will first be brought into the local address space (when needed, see Section 4.3.3) and then they are paged in by the above procedures.

# Chapter 5

# Protection

By protection we mean object access control, that is implemented in a integral, flexible and efficient manner. The mechanism we use was explained at several places in the early chapters. In the following we give an overview of it and show its integrity, flexibility and efficiency.

## 5.1 Definitions

- **Primitive access rights** are the basic access rights that one may have to an object. For example - read, write, copy, destroy, unseal.

- **Access rights** (AR) are a set of primitive access rights to an object.

- The **module's requested access rights** (RSAR) for an object are the access rights requested by the programmer for accessing the object.

- The **instruction required access rights** (IRAR) of an instruction to an object are the access rights it needs in order to act on that object.

- The **object's required access rights** (ORAR) of a program module to an object is a triplet <UID,MXRAR,MNRAR>, where UID is the object's identifier and MXRAR and MNRAR are the maximum and the minimum access rights needed by the module for accessing the object, taken over all the execution paths of the module. They are called **maximum required access rights** and **minimum required access rights**.

- The **module's required access rights** (MRAR) is the set of the object's required access rights of the module to all the objects it accesses.

- The **module's required capabilities** (MRC) of a program module is a set of capabilities for type managers, that are required at activation time of that module for extending the access rights of the module beyond those it may get normaly. This is called amplification, and usually involve the unsealing of certain objects.

## 5.2 The Protection Model

Our protection model deals with protection in two orthogonal dimensions. The first is the protection within a single process, which means the implementation of the minimum privilege to the modules that the process uses. The second is the protection between processes, that may be sharing program modules and objects. In fact, the model has two separate parts, that are connected at discrete points.

In both parts we try to verify access rights as early as possible. At compile (and link) time it is possible to know the access rights required for each primitive operation performed on an object. Yet, in general we do not know whether this operation is going to be executed at a specific run. In extreme cases we can know that, or we can know that the required access rights are to be granted in any case if the program module is going to be executed, since they are a subset of the requested access rights to the object. In other cases we expect the compiler and linker to provide the runtime system with the best information for detecting protection violation or deciding on inhibiting access rights verification as early as possible.

### 5.2.1 Intra Process Protection

Let us look at the flow graph of a process, where the nodes are program modules and the directed edges represent the flow of control between these modules. We do not require the program modules to be procedures (though in our protection mechanism we require them to be either instructions or procedures). They may be any piece of code.

Each node has its possessed access rights (PAR) to an object, which are time dependent, and include a list of objects and their access rights. We do not describe here how the node got its PAR, since this is part of the implementation.

The compiler finds the module's required access rights (MRAR) of each of the nodes. Dynamic objects that are added to the module's address space while the module is executed, will have a "delay" bit set in their MRAR entry. If there are requested access rights specified (RSAR) for any object, it will compare MXRAR to RSAR. If RSAR$\subset$MXRAR it does not generate object module and marks it as an error. If RSAR$\supseteq$MXRAR it will assign RSAR as the MRAR of the module. In any case, the compiler puts the MRAR in the object module.

The system deals with the access rights passed to a process by its creator, which is described later.

At run time, each time a node is entered, its possessed access rights

(PAR) are compared to its MRAR, excluding "delayed" MRAR entries. These are compared to PAR at the time they are added to the module's address space. In both cases, if PAR$\supseteq$MXRAR access rights verification is disabled for this activation of the node. If PAR$\subset$MNRAR then entry to the node is denied and the program is aborted. If none of the above then entry to the node is permitted but access rights verification is enabled for submodules of that node. Note that for a single instruction module, MXRAR=MNRAR therefore the last condition can never hold, and the above reduces to the condition that the instruction is executed if and only if PAR$\supseteq$MXRAR.

The above model may be implemented in any number of levels, including the single level, where each single access is verified at run time. The tradeoff is the classical one between execution time and space. The higher the number of levels is, the less we need to call on the verification mechanism and the earlier we detect protection violations. On the other hand, connecting MRAR to each node at each level takes a lot of space.

Note that in this model executing a node does not require more access rights then it actually need, except in the case where the programmer requests it.

### 5.2.2 Inter Process Protection

The above flow graph is related to a single process. Now let us deal with the protection model of several processes that may share different kinds of objects. This model is represented too by a flow graph, yet here the nodes represent processes and the edges are going from a creator to its sons. The link points between this model and the former are the *main* modules of each process, which actually represent the process in both models.

As before, the compiler finds the MRARs of the main modules of a process (node). At process creation time, the system will compare the MRAR (excluding "delayed" entries) of the main module of the process to the access rights that were granted to it by the process that created it (PAR). If PAR$\supseteq$MNRAR for all objects in MRAR, then the process is created and activated. Otherwise it is aborted. After the process is activated, if PAR$\supseteq$MXRAR for some object, then access rights validation for that object is disabled for the main module. Objects with delayed MRAR entries are treated the same as in Section 5.2.1.

## 5.3 The Protection Mechanism

The protection mechanism is capability based with typed memory, as in [Browne 82]. The mechanism is based on the model described in the earlier sections. We chose a two level implementation of the first part of the model, which is the nodes being protected procedures at the first level, and instructions at the second level. To this we add the second part of the model, that deals with processes. We think this implementation has a good time-space

balance, since protected procedures are big enough for the tables overhead to be relatively small, while granular enough to make MXRAR-MNRAR small. Since the only level where MXRAR always equal MNRAR is the instruction level, we must include this level too.

In the following sections we describe capabilities, then the mechanism itself, divided into sections according to their time and function. The times are program writing, compile time, process creation time, domain activation time and access time. The functions are data collection and verification. By data collection we mean finding RSAR, MRAR and MRC.

### 5.3.1 Capabilities

A capability consists of a UID and an access rights object. An access rights object will include two fields: a *deferred-bit* (DF) and an access rights field (ARF). If DF=0, then ARF specify the actual access rights. If DF=1 then ARF is a UID of an access rights object, which has the same format, and a mask.

The above scheme, while being transparent to the user, gives a solution to the revocation problem, which does not prohibit the reuse of the indirection space, which is prohibited in some indirection solutions. Another advantage is that we may use the same access rights object for accessing a group of objects.

Because it is regular and explicit, indirection in this method can be easily implemented by hardware.

#### 5.3.1.1. Access Rights Resolution

Access rights resolution takes place at the time an NNT entry is generated from a capability (or EID) by the ENTNNT instruction. At that time the hardware follows the access rights indirection chain, ANDing all the masks, together with the final direct access rights of the last stage. The result is the *resolved access rights* (RAR) that is entered to the NNT.

Note that following the indirect access rights chain is possible only by using the ENTNNT instruction.

### 5.3.2 Finding RSAR

The programmer may specify its requested access rights to certain objects. The requested access rights are a safeguard against programming errors that involve accessing objects in a way that is not intended. They may not be the access rights that the program is going to have, but they are an upper limit.

The requested access rights to an object may be specified to the compiler by some language constructs that will have the general form:

access object_name with ar1,...,arN;

where ar1...arN are the *symbolic access rights* (like read, write etc.) that the programmer need to be granted to him. The symbolic access rights are compiled to a bit pattern, where each bit correspond to one of the symbols. This bit pattern is the *module's requested access rights.*

### 5.3.3 Compile Time

At compile time we may do a modest amount of verification and a lot of data collection.

#### 5.3.3.1. Verification

If the object is internal to the compiled module, the requested access rights (if there are any) are the possessed access rights to the object and are verified by the compiler. If the object is external to the compiled module, the compiler assumes for the time being that the requested access rights are granted. In both cases, it verifies that all accesses to the object are consistent with the requested access rights, by consulting the object's type manager's *access rights table* (see Section 3.5.1). That means that all the access rights needed by the function performed on the object are included in the requested access rights. If some access violates the access rights, the compiler will mark them as errors and will not generate an object module.

#### 5.3.3.2. Finding MRAR and MRC

If there are no compilation errors, the compiler generates the *module's required access rights* table for each procedure in the object module. This is a list of the minimum and maximum access rights required by the procedure, that is used in the verification at linking or activation time. It also generates the *required capabilities* list for each procedure. This is the list of the capabilities for type managers that the procedure needs before it is allowed to run, in order to unseal certain objects.

The required access rights list include entries for all the objects accessed by the procedure, that are external to the compiled module. Its

entries are bit patterns that are calculated by the compiler for each object, by taking the OR of all the *access rights table* entries of the object's type manager functions that the module uses in certain path, then ORing over all the paths to get MXRAR, and ANDing over all the paths to get MNRAR. Each protected procedure in a domain has its own MRAR table. Its entries are parallel to the static NNT for static objects, and to the dynamic NNT skeleton for dynamic objects. The compiler sets the "delay" bit of entries for objects whose NNT entry is generated after the procedure entry.

### 5.3.4 Process Creation Time

At Process creation time we first find the access rights possessed by the process (PAR), then we verify them according to the data received from the compiler.

#### 5.3.4.1. Finding The Process' PAR

For a **job**, the system will find these rights as follows. The system uses the hardware instruction SYMTCAP that looks up a symbol in a context table. At the same time it looks at the principal's (user's) profile table for its permanent capabilities, and at the context table for the group access rights to the object, of the users group that the user belongs to (a user may belong to several groups). That means that the context table will contain a list of object names, their corresponding UIDs and their groups access rights. The possessed access rights for an object will be the OR of all the access rights of the groups that the user belongs to, together with those from its profile table. Dynamic objects whose NNT entry is created after the process is created, will also have a PAR entry generated for them. At the time their NNT entry is created, this PAR entry is ORed with the access rights that they may get in any other way.

Process creator other than the system may pass to the new process any subset of its own rights as PAR. Objects that are not included in this subset but have MRAR entries, will have empty PAR entries.

#### 5.3.4.2. Verification

Access rights verification are essentially the same as described in Section 5.2.2.

### 5.3.5 Procedure Activation Time

Here again, we first have to find the access rights possessed by the procedure, then verify the accesses.

#### 5.3.5.1. Finding The Procedure's PAR

A procedure possesses access rights to an object at a certain time, if it got them with a parameter when it was called, in a message through a logical channel or by having a type manager capability to the object's type.

#### 5.3.5.2. Verification

Verification is as described in Section 5.2.1, where adding an object into the module's address space means creating an NNT entry for it.

### 5.3.6 Access Time Verification

At access time we do only verification, since it is too late to collect data. At this point the access rights for the object are already resolved. The resolved access rights, which are the possessed access rights of the instruction node, are compared to the required access rights that are known to the type manager. The process is aborted if the resolved access rights does not include all the required access rights.

## 5.4 Flexibility

The flexibility of the protection subsystem is defined by the ease of access rights granting, revocation and amplification, without sacrificing the integrity.

### 5.4.1 Access Rights Granting

Granting of access right may be done in one of the following ways:

- Getting them in a capability from a procedure (type manager) as a returned value.
- Getting them in a capability from another process, through a logical channel or shared data segment.
- Getting then in a parameter.
- Amplification of access rights.

The first three ways may be used by the operating system to grant access rights to a job, after looking up in the system tables for access rights that were granted to the principal by one of the above methods. Basically, the source of all access rights to an object are its creator and its type manager. Together they share all the access rights possible to the object, and each may deliver parts of its share to others. This, again, is done by the above methods inside the system, but may be done by some methods outside the system, which we are not aware of and will assume the are trustable.

### 5.4.2 Access Rights Revocation

A process that likes to grant access rights to another process while reserving the privilege to revoke or restrict them later, will use the INDAR instruction, that accepts a capability (with direct or indirect access rights in it) and returns an access rights **object** and a new capability with indirect access rights, directed to the access rights object. The access rights object contains the original access rights field that was in the capability, and the new capability contains in its access rights field a pair <UID,mask>, where UID identifies the generated access rights object. The mask may be used to forther limit the access rights, and initially is all set. The access rights object, whether it contain direct access rights or <UID,mask> pair, may be restricted at any time, but not amplified.

The separation of the access rights object from the object's UID allows a group of capabilities for different objects to point to the same access rights object, and be revoked together. It also allows the name resolution process to be carried in parallel to the access verification, since the UID is known from the beginning.

From the access rights resolution process (Section 5.3.1.1) it is clear that anyone that owns a link on the indirection chain, may restrict the access rights at any time, and may revoke them altogether.

### 5.4.3 Access Rights Amplification

A procedure may amplify the access rights of some of its parameters for its own operation, if it has a special capability for the type manager of the parameter. The formal parameter access rights must be a subset of the access rights included in that capability for the parameter's type manager. The creator of the formal parameter list (and the procedure) gives this capability in the ENTNNT instruction when he creates that entry. Usually a creator of an abstract type (i.e. a type manager) will get such capability returned by the CREATE instruction, with all the possible access rights.

## 5.5 Integrity

By integrity we mean that no entity can access an object in a mode that was not legally granted to it (see Section 5.4.1) by an authorized entity. An authorized entity is an entity that legally possesses the access rights that it grants and has the right to grant them (this is a recursive definition). The integrity of the system therefore lays upon the integrity of the source of authority, the access rights granting scheme and the unforgeability of access rights.

The source of authority for accessing an object is shared between its creator and the type manager of the object. After creating the object, the type manager returns a capability for it with all the access rights accept the "unseal" right (to generate selectors for the object's elements) and the "amplify" right. No one but the creator will get access rights from the type manager. The creator (that may be the system) can then grant some access rights to others.

The access rights granting scheme is safe in the sense that no one is granted more rights than he is entitled to.

- When passing a capability, the grantor can restrict access rights to the exact amount that he wishes to grant. He can inhibit the ability of the receiver to make copies of the capability and send it to another process. The ENTNNT instruction destroys its operand capability after it uses it to open an entry in NNT. That means that a capability with no copy right that is received as a message can only be used directly from the input queue for creating an NNT entry.

- When passing a parameter, the procedure caller can restrict the access rights in the actual parameters table (see Section 3.4.1), except for amplification.

- Amplification is possible only by a procedure that has a capability for the parameter's type manager, with the "amplify" right. It can get it only from the type manager, being its agent or the type manager itself. Since the type manager is the other source of authorization, this is legitimate.

Forging access rights is not possible, because they appear only in capabilities, in access rights objects and in actual parameters tables, which are objects that can be either copied without change, or their access rights be decremented.

## 5.6 Efficiency

As in the case of addressing, the efficiency of the access control mechanism stems from the following:

- Early binding, which means in this case prerun verification of static accesses, that need not be checked later at run time, and resolving access rights at domain entry whenever it is possible.

- Use of the locality property of a program. This comes to use by the definition of protected domains, whose objects are protected from the outside world, but no run time access control is applied to local

accesses. Access control to local objects is provided by the compiler and linker.

- Use of capabilities frees us from the need to search access lists. The revocation problem is solved, though in the expense of extra indirections.

# Chapter 6

# Concurrency Control

By concurrency control we mean the implementation of certain policies to handle concurrent access to a shared object by several actors that may be ignorant of each other. These policies are meant to preserve the consistency of the system, which is the validity of certain relations between certain objects at certain points of the execution. We assume that the policy is defined in some specification language.

In the following sections we will discuss the existing methods for detecting and handling concurrency by the prerun software and by the run time system, and then give the TOBS's run time primitives that will be used for implementing these methods.

## 6.1 Discussion

Concurrency control is divided into the detection of concurrency and its elimination when it is not desired. Whether it should be eliminated and how is a matter of policy, which we will not discuss here, yet we will provide some mechanisms for implementing that policy.

The mere fact that several processes access the same object does not

necessarily mean concurrency, since they may be synchronized in such a way that concurrency does not occur. Certain kinds of concurrency are permitted (like reading the same object by several processes). If we find that we got one of these cases, we do not have to do anything about it. Otherwise, if we find that there is unpermissible concurrency or if we are not sure that it cannot occur, we must add more synchronization to the system. This may be done in one of two ways. One is by direct communication between the processes involved, which we call static concurrency control, and the other is by a common system device, which we call dynamic concurrency control. Although both have run time overhead, in many cases static concurrency control result in less overhead than dynamic concurrency control, since it can be adjusted to the exact amount of synchronization needed in the specific case. For example, synchronization at one point may preclude concurrency for many objects and accesses. In some cases static concurrency control has relatively high overhead, and in other cases it cannot be used at all, since we do not know in advance which processes are going to access the object. In such cases we must use dynamic concurrency control. There are also cases where we can use static concurrency control but the result may be a significant restriction of permissible concurrency and parallelism, which may decrease resource utilization and therefore throughput. In these cases we may prefer the use of dynamic concurrency control.

We may summarize the above by saying that we like to eliminate all illegal concurrency while minimizing run time concurrency control without interfering whith legal concurrency. A good reference on what may be done at compile time is [Richardson 84].

In the following we discuss the methods for detecting concurrency and eliminating it (when not legal), in both static and dynamic cases.

### 6.1.1 Concurrency Detection

We like to detect as much concurrency as we can at compile and link time, to be able to identify those cases that need no concurrency control, or decide on which concurrency control mechanism is appropriate. Unfortunately early detection is not always possible.

#### 6.1.1.1. Early Detection

By analyzing program behavior we can detect in some cases the inherent synchronization in a program and find the points where unpermitted concurrency may arise. This can be done by the compiler and linker using flow graphs [Taylor 83] for the case of synchronous communication, or by using a communicating finite state machines model and its reachability tree (when it is finite) in some more general cases. It should be noted that the general case where communication is unbounded is undecidable, but in practical cases we can put a bound on the communication (which may be the size of our queues) and get a decidable, though maybe not computable, case. That means practically that we do our best by the above methods, and whenever we cannot be sure, we assume that unpermitted concurrency may occur. For some stand alone systems (like ADA programs), we may detect much at compile and link time.

#### 6.1.1.2. Late Detection

By late detection we mean detection of unpermitted concurrency that was left by the early detection phase, either because it was not able to decide on it, or because it decided it will be better handled dynamically.

Late detection usually uses a busy flag that is set when the object is in use, thus allowing the system to detect more than one request for the same object. Usually the detection mechanism is combined with the control mechanism, like it is done in [HEP 82], where a tag is assigned to each of the active objects. The tag tells whether the object is empty or not, and may be used as a simple locking mechanism, that blocks certain accesses to an object while it is at certain state. This concept may be generalized to accommodate for more elaborate concurrency control mechanism, that may be used to implement different policies. Moreover, the mechanism may be applied not at each individual access, but rather at a procedure entry.

### 6.1.2 Concurrency Elimination

Concurrency elimination means synchronization of accesses so that concurrency does not occur. This can be done statically or dynamically.

#### 6.1.2.1. Static Concurrency Elimination

Static concurrency elimination is done by the compiler and linker by inserting synchronization points into the program. This can be done by any available mean of inter process communication.

A synchronization point should be inserted when:

1. The possible concurrency is early detected.

2. The other processes that share access to the object are known.

3. Parallelism is not seriously restricted as a result of the inserted synchronization point.

4. The added run time overhead is estimated to be less than the dynamic concurrency elimination run time overhead.

The second requirement seems to limit static concurrency elimination to a restricted group of programs, of which stand alone programs are a special case.

The third requirement can be met in general only if the synchronization mechanism does not impose ordering that is not needed otherwise. That means a mutual exclusion mechanism similar to [Ricart 81], which has an overhead proportional to the number of processes.

Fulfillment of the fourth requirement depends on the number N of processes involved (see last paragraph) and the expected number of concurrent accesses that are eliminated by each use of the inserted synchronization point. That means that static concurrency elimination may be useful when the number of processes involved is not big and the accesses to the shared object are concentrated in a small critical section that may be controlled by a single synchronization point without significantly restricting desired parallelism.

#### 6.1.2.2. Dynamic Concurrency Elimination

As mentioned in Section 6.1.1.2, dynamic concurrency elimination is combined with the concurrency detection. The busy flag is used as a lock that is set by one process to inhibit other processes access the object. The testing and setting of that flag is done by the the system, with at least some hardware support that prevents concurrent access to the flag.

## 6.2 TOBS's Concurrency Control

In TOBS we use the methods mentioned before for early and late concurrency control. in this section we describe only the runtime (late) concurrency control of TOBS.

### 6.2.1 Overhead

To reduce the overhead of the runtime concurrency control we use the following techniques:

- It is implemented in the A-processor microprogram and is conducted in parallel with the calculations in the E-processor.

- It is not applied to local objects, since it is assumed that a protected domain will take care of its own concurrency control, if there is any concurrency. Note that the protected domain itself is a protected object and runtime concurrency mechanisms may be applied to accessing it as a whole.

- It is activated only when an external pointer NNT entry opened or closed. That means that the object is locked as long as it has an entry in an active NNT, and the concurrency control mechanism is not applied to every single access. To avoid unnecessary locking, we may pass parameters as pointers to capabilities (or capability lists) and by that delay their locking until they are first needed.

### 6.2.2 The Mechanism

Each active object in TOBS has a *lock* that is attached to it at the time it is activated. The lock is part of the AOT table (see Section 4.2.3). In order to access an object, we also need a *key* and a *test mask*.

The *lock* will be implemented as a bit pattern that includes a field for each kind of access possible to that object. Each field will be treated as a counter of one or more bits. The high order bit of the field will be the actual lock-bit.

The *key* has the same structure as the lock - a field for each possible access mode. Fields that stand for access modes that are used in the type manager's function used to access the object will have non-zero values (see Section 3.5.2). Beside these fields a key has a *disabled* bit which, when set, disables the run time concurrency control checking on the object.

The *test mask* has the same structure as the lock. It has 1 in some of the MSB of its fields, depending on the type manager function used to access the object.

When no run time concurrency can be expected for an object, which means that it cannot be accessed from outside the linked module, the compiler and linker disable the dynamic concurrency control mechanism, by setting the disable bit in the concurrency key that it puts in the NNT (see next).

Each protected procedure has two *keys and test masks* tables, one which is parallel to the static NNT and the other which is parallel to the dynamic NNT skeleton. If run time concurrency can be expected for some object, the compiler generates a key and a test mask in a one of these tables depending on whether the object is static or dynamic. The key has in each field the maximum of all the corresponding fields of the keys required for each access to the object, as detected by the compiler. The test mask is the OR of all the test masks required for each access to the object.

When a protected domain that accesses a static non local object is

entered, and the key found in the table parallel to the static NNT object's entry is enabled, is is applied to its lock and the result is tested by the use of the corresponding test mask. If the object is dynamic, the key and mask are taken from the table parallel to the dynamic NNT skeleton and the test takes place when the dynamic NNT entry is activated. This process is done by the *concurrency controller* in the A-processor.

If any of the objects is already locked for the required access, the procedure will not be entered and the process will be put on the wait queue.

Applying a key to a lock is done by the CALL, RETURN and ENTNNT instructions, by means of the LOCK and UNLOCK operations of the concurrency controller. The LOCK operation will test the lock using the test mask TM. If the object is not locked for the requested kind of accesses, the key is added to the lock. In this operation, carry is not propagated between fields of the lock. Following is a formal description of the LOCK and UNLOCK operations.

```
LOCK(d,x,key):
   if d=1
   then begin
      if (lock(x) and TM(T(x))) = 0
         then lock(x):=lock(x) ++ key
         else perform an exception procedure
   end
   else begin
      if (lock(x) and ~TM(T(x))) ≠ 0
         then lock(x):=lock(x) - key
         else perform an exception procedure
   end

UNLOCK(d,x,key):
   if d=1
      then lock(x):=lock(x) - key
      else lock(x):=lock(x) + key
```

Where

| | |
|---|---|
| d | is the direction of locking (incrementing or decrementing). |
| x | is the object. |
| T(x) | is the object's type. |
| ++ | means adding without carry propagation from one group to another. |

The above may be implemented as:

```
LOCK(d,x,key):
   if [(xor(d,TM(T(x))) and lock(x)) ≠ 0] =d
   then lock(x):=lock(x) ++ xor(d,key)+d;

UNLOCK(d,x,key):
   lock(x):=lock(x) ++ nxor(d,key)+~d
```

The LOCK and UNLOCK primitives are similar to the *fetch and add*

operation described in [Gottlieb1 83, Gottlieb 83], except that we suggest the addition to be conditional. By that we make it more like a generalized *test and set* operation and avoid the need for extra test.

## 6.3 Examples

The following examples are similar to those given in [Gottlieb1 83], except that they use our primitives. We also give an example for the use of a multiple lock.

### 6.3.1 Semaphores

Semaphores are implemented by a lock and a key having a single bit. The lock is initiated to 0, while the key masks TM and LM are both set to 1. The object s may be a small object that is used only for its lock, but more likely it will be the object that is protected by the semaphore. That may be a procedure that contains a critical section or some data object. We define:

```
P(s) = LOCK(1,s,key)
V(s) = UNLOCK(1,s,key)
```

We may combine several semaphores in a single lock and LOCK them at the same time. This may be found useful for avoiding deadlocks if the semaphores belong to different objects, which they do not. We mat, though, add the possibility of putting a pointer in key, so it may point to different locks. That should by worked on forther.

### 6.3.2 Readers-Writers

We assume that up to $N=2^n$ simultaneous readers and no writers, or one writer and no readers, are allowed to access the object at any time. To implement this we use a lock with n bits initiated to 0, a read key set to 1 and a write key set to N. The read and write access mode procedure will be as follows:

```
procedure Reader;
LOCK(1,s,Rkey);
read-body;
UNLOCK(1,lock,Rkey);

procedure Writer;
LOCK(1,lock,Wkey)
write-body;
UNLOCK(1,lock,Wkey);
```

### 6.3.3 Multiple Lock Example

Let us simulate a bus with N seats that may travel in one of k routes. A traveler that wants to travel on route i will apply access mode $m_i$ to the bus. If the bus is empty, the driver will let the traveler on and set the bus route to be route i. That means that it will not let travelers on, unless they need the same route and as long as there is still space on the bus. That is, they have to apply access mode $m_i$, and the number of passengers on the bus should be less than N.

The simulation will be easy if $N=2^n$ for some n. In this case we will have a key with k groups, each of n+1 bits. TM will have a 1-bit at the most significant bit of each group, while the rest of the bits will be zero. LM will have a 1-bit at the least significant bit of each group, while the rest of the bits will be zero.

The access-mode procedure will apply LOCK(0,x,key) at the beginning of the access (ride), and UNLOCK(0,x,key) at the end of the access (if it was granted).

# Chapter 7

# Parameterization of the Architecture

In the following we assign values to the system's parameters, to be used in the evaluation. One of the considerations for assigning these values is the similarity to other systems, so the evaluation is done on a fair basis.

## 7.1 Memory Parameters

Token 4 bits.

Page size 1024 tokens.

Main memory 1MB (= 2M tokens).

## 7.2 capabilities

Capability 68 bits if direct, 116 bit if deferred:

Type code 4 bits

Access rights object 16-64 bits:

Deferred 1 bit.

Mask disable, read, write, execute, copy, unseal, destroy, locality, amplify, special. (15 bits).

Deferred UID 48 bits (if any).

UID 48 bits

## 7.3 Pointers

Short 12 bits (A-field size=4 bits).

Medium 20 bits (A-field size=11 bits).

Long 28 bits (A-field size=19 bits).

A field parameters (see Section 3.2.1.2):

RN 4 bits.

RS 4 bits.

## 7.4 O-pointer

128 bits.

## 7.5 P-pointer

36 bits.

## 7.6 Addressing Tables

GOT 10000 entries, 128 bits each:

UID 48 bits.

Object address 32 bits.

Object length 32 bits.

TRC 16 bits.

AOT 2048 entries, 128 bits each:

NEXT 12 bits.

BACK 12 bits

UID 48 bits.

PF 22 bits.

D 10 bits

ARC 8 bits.

LOCK 16 bits.

NNT 16 to 2**11 entries, 128 bits each. Selectors use two consecutive entries. Parameters entries use three consecutive entries. Entries description:

- Empty entry:

  Code=0 2 bits.

- External pointer:

  Code=1 2 bits.

  LCL 1 bit.

  Spare 1 bit.

  TP 8 bits.

  SONS 12 bits. Beginning of sons list.

| | |
|---|---|
| RAR | 16 bits. |
| UID | 48 bits. |
| AOP | 12 bits. |
| DD | 12 bits. Offset in first page. |
| PN0 | 16 bits. |

- Selector:

| | |
|---|---|
| Code=2 | 2 bits. |
| LCL | 1 bit. |
| Spare | 1 bit. |
| TP | 8 bits. |
| SONS | 12 bits. Beginning of sons list. |
| AM | 12 bits (primitive or relative NNT index). |
| F | 12 bits. |
| EF | 12 bits. |
| EN | 24 bits. |
| SIZE | 24 bits. |
| DD | 12 bits. |
| NXBR | 12 bits. Next brother. |
| OBP | 32 bits. |
| Ptag | 16 bits. |

| | |
|---|---|
| P1 | 24 bits. |
| P2 | 24 bits. |
| P3 | 24 bits. |
| Lock | 4 bits. |

- Parameter entry (256 bits):

| | |
|---|---|
| Code=2 | 2 bits. |
| LCL | 1 bit. |
| Spare | 1 bit. |
| TP | 8 bits. |
| SONS | 12 bits. Beginning of sons list. |
| Spare | 12 bits. |
| F | 12 bits. |
| EF | 12 bits. |
| EN | 24 bits. |
| SIZE | 32 bits. |
| RAR | 16 bits. |

if LCL=0:

| | |
|---|---|
| OBP | 32 bits. |
| Ptag | 16 bits. |
| FNNT | 48 bits.<br>The UID of the NNT where the father's entry resides. |

| | | |
|---|---|---|
| | EFNNT | 48 bits.<br>The UID of the NNT where the external father's entry resides. |
| | Otherwise: | |
| | OPAR | 128 bits. O-pointer of the parameter. |

RPT 2048 entries, 128 bits each.

APT 4096 entries, 128 bits each.

RAT 64 entries, 24 bits each.

OPT head: 64 bits.

OPT entry 32 bits:

| | |
|---|---|
| PPF | 22 bits (permanent page frame number). |

## 7.7 Fields

PN 22 bits.

PF 22 bits.

PI 16 bits.

D 10 bits.

PP 12 bits.

OP 8 bits.

ARC 8 bits.

PVA 72 bits.

Character size
8 bits.

PTP 32 bits.

## 7.8 Object Size

Many of the object types has variable size. In some cases (like array) the size is not given directly but is implied by other parameters. In other cases there is a special size field. Object size field itself has a variable size. Except for pointers, that are described in Section 7.3, we basically adopted the method used in [Browne 82], but we modified it in the favor of small objects. The first 1-4 bits has the following format:

0000 2 data tokens.

0001 4 data tokens.

0010 8 data tokens.

0011 16 data tokens.

else Browne and Smith size (see [Browne 82]).

## 7.9 Fast Memories

Operand cache
: 1024 X 32 tokens.

AGV fast memory
: 256K tokens.

Inst. lookahead
: 256 tokens.

SR
: (Shift Register) 2 X 64 bits.

## 7.10 Internal Registers

The A-processor will have 32 internal registers, each 64 bits wide. It will have a 128 bit shifter, so bits can be shifted from one register to the other.

The E-processor will have 8 64-bit internal registers and the arithmetic will be 64-bit parallel.

## 7.11 Opcodes

Counting bit 0 to be the most significant bit of the opcode, we distinguish the following bits:

OP[0]
: 0: E-processor short (4 bits) opcode (SOP).
1: long (8 bits) opcode (LOP).

LOP[1]
: 0: E-processor opcode.
1: A-processor opcode.

OP[2] 0: no operands.
1: one or more operands.

### 7.11.1 E-processor Short Opcodes

The following are 4 bit (one hexadecimal digit) opcodes.

0 PUSH

1 POP

2 PUSH z.

3 POP z.

4 EXCH

5 NOP

6 PUT z

7 GET z

### 7.11.2 E-processor Long Opcodes

The following are 8 (two hexadecimal digits) bit opcodes.

80 ADD, OR

81 SUB, XOR

82 CMP

83 MUL, AND

| | |
|---|---|
| 84 | DIV, SHIFT |
| 85 | NEG, NOT |
| 86 | INC |
| 87 | DEC, TRUE |
| 88 | CLR, FALSE |
| 89 | spare |
| 8A | Ereset E-processor. |
| 8B | OPQclear. |
| 8C | EAQclear. |
| 8D | CLRS |
| 8E | spare. |
| 8F | spare. |

### 7.11.3 A-processor Opcodes

The following are also 8 bit opcodes.

| | |
|---|---|
| E0 | AADD, STEP |
| E1 | ASUB |
| E2 | MOV, SRESET, SEND, RECEIVE |

| | |
|---|---|
| E3 | COPY, CHTEST |
| E4 | ASHIFT |
| E5 | ADEC |
| E6 | AINC, NEXT |
| E7 | DBNZ |
| E8 | ACLR, CLEAR |
| E9 | CHDSEG |
| EA | ENTNNT |
| EB | NXB |
| EC | NEWPET |
| ED | INADR |
| EE | CREATE, CONNECT |
| EF | DESTROY |
| F0 | ACMP |
| F1 | SET |
| F2 | CONVERT, SYMTCAP, UIDTCAP |

| | |
|---|---|
| F3 | REP |
| F4 | EXECUTE |
| F5 | BA (branch according to A-processor result) |
| F6 | BE (branch according to E-processor result) |
| F7 | CALL |
| D7 | RETURN |
| F8 | ENTRY |
| F9 | WAIT |
| D9 | EXIT |
| FA | |
| FB | SWITCH |
| FC | RSWITCH |
| FD | RESPNT |
| FE | BRINGP |
| FF | STOP |

# PART II: EVALUATION

# Chapter 8

# Analysis

Performance analysis is a major element in the design of any new computer architecture. Performance analysis is particularly important for object-oriented architectures since they have often had performance deficiencies. The performance analyses reported herein were aimed at evaluating the effectiveness of the TOBS solutions for the performance deficiencies of previous object-oriented systems.

There are two aspects to the analysis and evaluation. One is to determine the relative effectiveness of the several components of the TOBS architecture. This information will be used in improving the design of future versions of TOBS.

TOBS is aimed at bringing the concepts of object-oriented architecture to the micro-processor execution environment. Therefore the other aspect of the evaluation is comparison to a standard micro-processor of similar complexity. The Motorola 68000 was selected for comparison.

The analyses and evaluations were based upon the execution of several small programs which were designed to test critical areas of

architectural design including test loop control, indexing, and procedure calls as well as arithmetic. These operations have been the source of performance problems in previous object-oriented systems.

The programs included linear search, recursive quicksort, vector dot product and matrix product. The programs were written in assembly language for both TOBS and the Motorola 68000.

The TOBS programs were executed on a register level simulator. The 68000 programs were run on a VALID Logic workstation.

A number of metrics were gathered for each execution including total clock cycles to complete the execution and the number of memory references. The clock cycle count was chosen as being the most comparable metric across architectures with different instruction sets. Memory reference counts are a metric for the effectiveness of caching and localities mechanisms in the architecture.

The next chapter defines and describes the TOBS assembler and simulator, gives the programs, displays the performance results and analyses the evaluation they provide on the TOBS architecture.

# Chapter 9

# Simulation

Simulation was done by a simulator that simulates TOBS at the functional unit and register level. Time simulation is based on clock cycle counting. The simulator is written in C and executes under the Unix operating system. It includes an assembler with some special language constructs that allow the definition of various objects and type managers, and an interactive runtime system that simulates the functions of the hardware and some of the OS functions.

## 9.1 The Assembler

The assembler is a two pass assembler that translates assembly language directly into a runnable program module. The program includes one or more domains, one of which is the main module where execution initiates. The assembler is intended solely as a tool for writing the benchmark programs that are used for our design evaluation. Therefore certain assumptions were made in order to keep the assembler simple.

It is assumed that all the components of a domain are present at assembly time, so no linking is needed at any later time. Yet, separately assembled modules can be called through the remote procedure call mechanism.

Since the assembler has only two passes, wherever the size of a pointer cannot be determined in the first pass, it is assumed to require a long address field. This relates to branch instructions with forward references and to pointers in a domain's tag.

The assembler provides the programmer with tools to define abstract types, domains, access rights and other objects that on the real system will be defined by the compiler and not directly by the programmer. That means that the programmer must do part of the job of the compiler and must therefore do it carefully.

### 9.1.1 The Assembly Language

The program is defined as a stream of lines, each having the following format.

#### 9.1.1.1. The Input Line

The input line has a free format, with the following fields:

label: opcode operand1,...,operandN [; comment]

Fields that are not relevant need not appear. The opcode field is mandatory.

#### 9.1.1.2. Pseudo Instructions

- Domain Definition

```
name:  domain
       interface e1,...,eN
p1:    type
p2:    type
.
```

```
.
.
pnN:  type
      body
```

where:

| | |
|---|---|
| name | is the domain's name. |
| e1,...,eN | are labels defined in the domain's body and can be used in calls from outside the domain. |
| p1,...,pnN | are all the formal parameters used in the procedures e1,...,eN. They actually define the parameter section of the dynamic NNT skeleton. Reference to pk will be translated by the assembler to a dynamic segment reference with A-field of -k. The above may result with the need of actual dummy parameters in some cases. (That need may be eliminated by a smarter translator, but the dummy parameters will still appear in the object module). |
| body | is a block of code that does not contain domain or abstype instructions. |

If a domain is not declared at the beginning of the program, a default protected domain with the name "PROG" is opened at the beginning of the program. The domain ends (in both cases) with the beginning of the next domain, or with the end of the program.

- Abstract Type Definition

```
name:  typeman
       struct representation
       interface e1,...,eN
p1:    type
p2:    type
.
.
.
pnN:   type
```

body

where representation defines the objects of that type in terms of earlier defined types.

- Segmentation Instructions

These instructions define the segment where the code that follows should be entered. Their scope ends with the next segmentation instruction.

| | |
|---|---|
| own | starts a block of own (local data) variables. |
| static | starts a block of static (process' data) variables. |
| dynamic | starts a block of dynamic (local stack) variables. |
| global | starts a block of global (NNT) variables. |
| code | starts a block of code. |

- Storage Allocation

Local objects are always allocated storage in the appropriate segment when they are declared. Global objects may be allocated space by another module and we only declare them for their NNT entry, or else we may like to create both the object and its NNT entry. For that we have the following pseudo instructions, which are used to subdivide the global declarations into several sections:

| | |
|---|---|
| define | opens a definitions section in a global segment, where each declared object is allocated both an NNT entry and physical space. |
| locate | opens a section in a global segment, where each declared object is allocated only an NNT entry. The assembler (in real life - the linker) will locate the object that is defined (hopefully) in some other domain, or locally in the same domain, and resolve its address in the NNT entry. |

undef — opens a dynamic section in a global segment. The objects declared in this section are allocated NNT entries, but they are neither allocated space nor they are being located in the global segment.

The scope of these instructions is until the next storage allocation, segmentation, domain or type manager instruction.

- Data Generating Instructions

These instructions generate data objects of a certain type in one of the above segments. Each data element must be initialized, either by the programmer or by default to "undefined" value.

The instructions opcodes are essentially the type names, as follows:

int[size] — defines an integer with an optional initial value.

real[size] — defines a real with an optional initial value.

boolean[size] — defines a boolean with an optional initial value.

string[size] — defines a string with an optional initial value.

```
array[size] of
  type[size]

record
  type1[size]
  .
  .
  .
  typeN[size]
recend
```

pointer[size]

capability

*element* en *of* father *mode* name(p)

The instruction line has the following format:

name: type[size] value:<minAR,maxAR>

The element instruction is somewhat exceptional. en is the initial value (element number), father is the father entry name, and name(p) is the access-mode name with its parameter. Access mode names are as follows:

| | |
|---|---|
| direct | which is the default if access mode is not specified. Element number is explicitly set by the user for each access. |
| step | p specifies the step size, positive or negative. |
| list-step | step to the next element in a linked list. p may be 1 or -1. |
| extended | a name of a user procedure that calculates the next element's address. |

If size is not given, the size of the object defined on the last line is used, if it is of the same type. Otherwise a default size is used.

Value is the value to be entered as the object's value.

minAR and maxAR are the minimum and maximum access rights required for the object. For example:

{{read,write}{read,write,copy,unseal}}

If the object is not global, only one list should be used. The default access rights are the total access rights.

Example:

```
   global
   define
x: array[5] of real 5,4,3,2,1 : {{read}{read,write}}
y: element 3 of x mode step(2)
```

#### 9.1.1.3. Assembler Directives

EQU gives its label or name the value of its operand, which must be a label or a name, correspondingly (not an expression or constant).

CONST gives its label the value of its operand. The operand is an expression involving other earlier defined constant names and numerals, with the operators +,-,/,*.

END ends the program. Its operand is the program start label.

#### 9.1.1.4. Constants

A constant is an object that is given a value that is defined to the assembler, and cannot be changed. Constants may have symbolic names, that are resolved by the assembler, or their name may be their value, in which case we call them direct constants. Constants may be simple or structured. A structured constant is a structured object whose elements are constants.

We distinguish the following representations of direct constants:

decimal a base 10 integer.

octal a base 8 integer, preceded by at least one zero.

hexadecimal a base 16 integer, preceded by 0x.

fixed point a base 10 real number that includes a decimal point.

floating point a base 10 real number that includes the exponent symbol E and a power following it.

boolean a string of zeros and ones, preceded by 0B.

string a string of characters, enclosed by " or '.

#### 9.1.1.5. Addressing Modes

Following are the symbolic representations of the addressing modes of an instruction operand:

#object immediate mode.

May have one of two forms:

#constant (see Section 9.1.1.4) for simple objects.

#type{value}
for structured objects.

@pointer indirect mode.

nickname direct reference to a global (or structured local) object.

(nickname) a reference to the integer EN field of a selector.

Selecting the right segment is done automatically by the assembler, knowing where the object was defined.

### 9.1.2 The Assembler Output

The assembler and simulator will maintain a file "auxmemory" which is the image of the TOBS auxiliary memory. If such file does not exist, the assembler will create it. If it already exists, the assembler will read its bit map and use free pages to store the program. The auxiliary memory address range is $2^{22}$ to $2^{32-1}$ (while the main memory addresses range from 0 to $2^{22-1}$). At the end of the assembly the simulated auxiliary memory (auxmemory) will contain the program in binary, together with the GOT table and some system's pointers.

The assembler will allocate space in auxmemory for static global objects (like the protected domains and data objects), generate UIDs and create GOT and context table entries for them (the last generated UID is found in LASTUID object.

The system's pointers mentioned above are included in a system's table called "the system's root" that is built by the assembler at page 0 of auxmemory. The pointers in this table are 8 tokens P-pointers, and they point to the GOT and the system's context table.

Following is the description of the tables that the assembler uses and modifies.

### 9.1.2.1. Auxiliary memory

The format of "auxmemory" file is as follows:

Record size: 256 bytes.

record 0 — contains system's pointers, the file size (up to 4096 pages), creation date and date of last access.

record 1,3 — contain system's bootstrap loader.

record 4-35 — contains the memory availability pattern (MAP) of 32K bits, in which each 1 bit tells that the corresponding page (that includes 4 records) is free. That includes the first page, which is comprised of records 0-3 above. Bits are counted from least to most significant.

record 36-1059
1024 records of 256 page external AOT.

record 1060-1063
GOT's first page.

record 1064-1067
context table's first page.

record 1068-1071
principals table first page.

#### 9.1.2.2. Context table

The format of a context table entry is:

| | |
|---|---|
| symbolic name | 20 tokens |
| UID | 12 tokens |
| Group ID | 12 tokens |
| group AR | 4 tokens |
| others AR | 4 tokens |

#### 9.1.2.3. Profile Table

Is an object of type 'E3' that contains two pointers: a pointer to a job's characteristics table and a pointer to a capability list. The profile table length is 18 tokens.

#### 9.1.2.4. Characteristics Table

Is an object of type 'D1' that id contained in a single page and has the following format:

0: type code D1 - 2 tokens.
2: number of entries - 6 tokens.
8: job's default context table pointer - 8 tokens.
16: pointer to principal's profile table - 8 tokens.
24: principal's profile table's UID - 12 tokens.

#### 9.1.2.5. Principals Table

Is an object of type 'D2' that has the following format:

0: type code D2 - 2 tokens.
2: PTP - 8 tokens.
10: number of entries - 8 tokens.
18: principal's symbolic name - 20 tokens.
38: pointer to principal's profile table - 8 tokens.
46: principal's profile table's UID - 12 tokens.
58: next entry.

The assembler and the simulator get the user ID from the (Unix) system and use it to find the user's entry in the principals table, that points to its profile table and to its context table. The profile table includes, beside the pointer to the context table, a list of groups to which the user belongs.

The assembler also produces a listing file of the program, including a global symbol table (i.e. - the context table).

## 9.2 The Simulator

The simulator is written in C and executes under Unix operating system. It simulates all the units of TOBS, as they are described in Chapter 2. The simulation is at the register level at each unit. Communication between different units are through queues and buffer registers, as described in Chapter 2.

The simulator is the main tool for evaluation of TOBS design. It provides information on the TOBS program that it runs. That information includes the time it take to run the program, expressed as number of internal

clock cycles, the number of TOBS instructions actually executed and some cache and memory management statistics. An example of the simulator's output is given in Table 9-1.

### 9.2.1 Time Simulation

Since the simulator does not run at real time, real time has to be simulated. Furthermore, the simulator runs as a sequential process while TOBS has several units that may run in parallel, which complicates the simulation of time.

Time simulation is based on Lamport's clocks (see [Lamport 78]). Each unit has it own clock, which is advanced sequentially according to the activity that it performs. Each queue element or buffer register is time-stamped with the local time of the unit that access it, each time is is filled or emptied. When a unit wants to either put or get an element from a queue or a buffer register, it first looks at the time stamp. If it is less than its own clock value, it goes ahead. Otherwise it has the option to execute any valid operation that does not involve that queue or register, or to wait (by setting its own time to the time stamp of the element).

Execution inside each unit is sequential, but not in a predetermined order. To emulate a pipeline, it has a number of internal registers which hold intermediate results. Each register has a tag that tells whether it is full or empty, and additional information about its contents. The simulator searches the registers to find an action to execute. If it cannot find anything to do in its internal pipeline, it will look at the input and output queues and registers

of the unit. If the time stamps of the top elements of all these queues are of higher time than the unit's local clock, it will wait for the one with the lowest time stamp and then try to proceed internally. This process continue until no more progress can be made. At this point control is passed to the emulator of a unit with the lowest local time.

Units are called by one another in an hierarchical fashion, which makes it easy to detect a deadlock or termination. Each unit emulator returns a value that tells whether it has made any progress, and this is accounted in its caller's decision of whether it has made any progress.

### 9.2.2 Validation of The Simulation

The validation of the correct behavior of the simulator was done by extensive testing. A large number of test programs which test different parts of the simulator were run, and the results were validated by hand. The component tests included the object access subsystem, the paging subsystem, the cache and the unit emulators.

The most difficult validation was of the time calculation. This was done by running small examples that could be analyzed. Parallelism, pipelines and cache made analysis very hard. Parallelism, pipelines and cache were disabled during these tests. The time calculations were carefully examined to verify faithful implement of the algorithm for the time calculation, and are believed to be correct.

First run:
execution time: 558 cycles
Instruction executed: 18
Average instruction execution time: 31.111111 cycles

Number of memory token references:
cache: 2400 other: 1958 total: 4358

operand cache statistics:
# of reads 162 # read hits 129 %= 79.629631
# of writes 56 # write hits 48 %= 85.714287
# of page faults = 12 %= 11.320755

Second run:
Instruction executed: 407
Average instruction execution time: 6.744472 cycles

Number of memory token references:
cache: 21933 other: 3117 total: 25050

operand cache statistics:
# of reads 1729 # read hits 1685 %= 97.455177
# of writes 250 # write hits 242 %= 96.800003
# of page faults = 12 %= 1.359003

Third Run:

execution time: 23904 cycles
Instruction executed: 4007
Average instruction execution time: 5.966059 cycles

Number of memory token references:
cache: 203130 other: 14228 total: 217358

operand cache statistics:
# of reads 16268 # read hits 16111 %= 99.034912
# of writes 2056 # write hits 2048 %= 98.610893
# of page faults = 15 %= 0.185300

**Table 9-1:** Simulator output for the Linear Search

## 9.3 Testing

After the simulator became available, several programs were run on it and on the Motorola 68000, in order to compare their performance under different inputs. The Motorola 68000 was chosen because it is considered a good representative of conventional micro-processors and it has been compared to other machines (see [Hansen 82]), which makes it possible to carry the results of the comparison to these other machines.

The Motorola 68000 tests were run on a Valid Logic work station that includes a Motorola 68010 processor with 8MHz internal clock rate and one Wait state per memory access. Run times were obtained using a counter with a 10 microseconds accuracy, that was started and stopped by an output directed to a certain port. The results were averaged over 25 runs, so the effective accuracy is 0.4 microseconds or 3.2 clock cycles.

The description of the test programs that were run follows.

### 9.3.1 Linear Search

Search an integer key in an array of integers and return the index of the array element where the key was found, or -1. The program was written as a main program in one domain, that calls the search procedure in another domain. We ran the program with the search lengths (number of elements scanned until the key is found) of 3, 100 and 1000. The program and its results are listed in Figure 9-1 and Table 9-2.

This program is intended to test performance of a relatively small

loop. The results show that as the number of iterations increases, the average instruction execution time decreases asymptotically. That is explained by the overhead of the remote procedure call at the beginning of the program execution, and the overhead of initial resolution of certain names. Part of it may be related to initially loading memory pages from auxiliary memory, and loading the cache. These events repeat occur much less frequently later in the execution, as can be seen by the increasing cache hit rate and decreasing page fault rate.

### 9.3.2 Recursive Quick Sort

Sort an array of reals by the recursive quick sort algorithm. The program was written as a main program in one domain that calls the sort procedure in another domain. The recursive calls were local calls to the same procedure. The program was run with the array sizes of 3, 100 and 1000 elements. The numbers in the arrays were chosen randomly. The program and its results are listed in Figure 9-2 and Table 9-3.

### 9.3.3 Dot Product

Compute the dot product of two arrays of real numbers. Intended to test the effect of the parallelism between the A-processor and the E-processor, with heavy floating point processing and loop control. The program was written as a main program in one domain that calls the dot product procedure in another domain. It was run with array sizes of 20, 200 and 1000 elements.

To test the effect of using a selector, compared to use of a regular (static) variable, we run another version of the program, that multiplies an array by a scalar. The result is a reduction of more than 20% in the run time.

```
; Linear search program.
;
linsearch: domain
           interface    lsearch
;
        own
res:    integer
ary:    array[100] of integer
acp:    ACP     {<#ARF{RE},ary><#ARF{RE},#I{13}><#ARF{RE},#I{100}>
                 <#ARF{WRT},res>}
        global
        locate
linsearcha: edomain {}
        define
search: element 0 of linsearcha
;
        code
lsearch: entry  DYNSIZE
        call    search,acp
        stop
        end
```

**Figure 9-1:** The Linear Search Program

```
;
; Linear search procedure.
;
linsearcha: domain
            interface   search
ary:    array[] of integer  {{0x7fff}{0}}
key:    integer
size:   integer
result: integer
;
        global
        define
test:   element of ary
;
        code
search: entry   DYNSIZE
        set     test,#I{0},#I{1},#P{4,3}        ; set selector
        push    #P{4,2}          ; push key
top:    push                     ; duplicate key
        cmp     test
        be      EQ,found
        nxb     (test),top
        copy    #I{-1},#P{4,4}   ; not found
        pop
        return
found:  copy    (test),#P{4,4}
        pop
        return
        end
```

Figure 9-1, continued

TOBS results:

| run | search size | # of inst. | cycles | ave. inst. | r-hit rate | w-hit rate |
|---|---|---|---|---|---|---|
| 1 | 3 | 18 | 558 | 31.1 | 79.6% | 85.7% |
| 2 | 100 | 407 | 2743 | 6.74 | 97.4% | 96.8% |
| 3 | 1000 | 4007 | 23904 | 5.97 | 99.0% | 99.6% |

M68000 results:

| run | search size | cycles |
|---|---|---|
| 3 | 1000 | 58800 |

**Table 9-2:** Linear Search Test Results

```
;       testsort
;
; Test quick sort.
;
testsort: domain
         interface  start
;
        own
ary:    array[100] of real
left:   integer 0
right:  integer 100
acp:    ACP     {<#ARF{RE,WRT},ary><#ARF{RE},left><#ARF{RE},right>}
;
        global
        locate
rquick: edomain {}
        define
qsort:  element 0 of rquick
;
;
        code
start:  entry   DYNSIZE
        call    qsort,acp       ; use default access rights
        stop
        end


; Recursive quick sort.
;
rquick: domain
        interface  qsort
ary:    array[] of real {{0x7fff}{0}}
left:   integer
right:  integer
;
```

**Figure 9-2:** The Quick Sort Program

```
        global
        define
aryi:   element of ary
aryj:   element of ary
        dynamic
intj:   integer[10]
p1:     integer[10]
p2:     integer[10]
        own
l:      integer
r:      integer
acp:    ACP     {<#ARF{RE,WRT},ary><#ARF{RE},l><#ARF{RE},r>}
;
        code
qsort:  entry   DYNSIZE
        copy    #P{3,2},p1
        copy    #P{3,3},p2
        acmp    p1,p2
        ba      GE,qend          ; stop the recursion
        set     aryi,p1,#I{1},p2        ; set selector
        set     aryj,p2,#I{-1},p1       ; set selector
        push    aryi             ; push left element as key.
        ba      T,inci
l1:     push
        cmp     aryi
        be      GE,decj          ; until key <= ary[i]
inci:   nxb     (aryi),l1        ; repeat i:= i + 1
        ba      T,decj
l2:     push
        cmp     aryj
        be      LE,found         ; until key >= ary[j]
decj:   nxb     (aryj),l2        ; repeat j:= j - 1
;
```

Figure 9-2, continued

```
found:  acmp    (aryi),(aryj)
        ba      GE,endlp          ; if i >= j then
        push    aryi              ; swap ary[i] & ary[j]
        push    aryj
        pop     aryi
        pop     aryj
        ba      T,inci            ; continue until i<= j
;
endlp:  copy    p1,(aryi)         ; load aryi with left index
        push    aryj              ; swap ary[left] & ary[j]
        pop     aryi
        pop     aryj
;
        copy    (aryj),intj
;
        copy    intj,r
        ainc    r
        acmp    p1,intj
        ba      GE,call2
        acmp    p2,r
        ba      GT,call1
        copy    intj,r
call1:  copy    p1,l
        call    qsort,acp         ; use default access rights
;
call2:  ainc    intj
        copy    intj,l
        copy    p2,r
        call    qsort,acp
;
qend:   return
        end
```

Figure 9-2, concluded

TOBS results:

| run | array size | # of inst. | cycles | ave. inst. | r-hit rate | w-hit rate |
|---|---|---|---|---|---|---|
| 1 | 3 | 117 | 1945 | 16.64 | 93.9% | 87.6% |
| 2 | 100 | 8872 | 94515 | 10.65 | 99.8% | 99.4% |

M68000 results:

| run | array size | cycles |
|---|---|---|
| 2 | 100 | 140560 |

**Table 9-3:** Quick Sort Test Results

The program and its results are listed in Figure 9-3 and Table 9-4. We can recognize here a behavior similar to that of the first test.

### 9.3.4 Matrix Product

Compute the product of two large matrices of real numbers. It intends to test the effect of remote versus local parameter references, as well as to further test the effect of the A/E processor parallelism, with heavy floating point and loop control processing.

In order to test local versus remote parameter referencing, the program was written in two versions. One version was written as a main program in one domain that calls the matrix product procedure in another domain. The other version includes both the main program and the procedure in the same domain. The size of the matrices is 50 X 50, which involves a lot of processing, so that the dominant effect is that of the main loop, which includes the parameter referencing. Thus, the effects of the remote procedure call are not noticed.

The program and its results are listed in Figures 9-4, 9-5 and Tables 9-5,9-6. We can see that the difference in the results between the runs with the local and remote procedure calls is minor.

```
;       dotprod
;
; Large arrays dot product
;
dotprod: domain
         interface      prod
;
        own
size:   integer 1000
res:    real
ary1:   array[1000] of real
ary2:   array[1000] of real
acp:    ACP     {<#ARF{RE},ary1><#ARF{RE},ary2><#ARF{RE},size>
                 <#ARF{WRT},res>}
;
;
        global
        locate
dotproda: edomain {}
        define
dprod:  element 0 of dotproda
;
        code
prod: entry     DYNSIZE
        call    dprod,acp
        stop
        end
```

**Figure 9-3:** The Dot Product Program

```
; Dot product procedure
;
dotproda: domain
          interface     dprod
ary1:   array[] of real  {{0x7fff}{0}}
ary2:   array[] of real  {{0x7fff}{0}}
size:   integer
result: real
;
        global
        define
x:      element of ary1 mode step
y:      element of ary2
        own
zero:   real    0.0
        code
dprod:  entry   DYNSIZE
        set     x,#I{0},#I{1},#P{4,3}    ; set selector
        set     y,#I{0},#I{1},#P{4,3}    ; set selector
        push    zero
;
loop:   push    x       ; in version 2 this is a regular variable!
        mul     y
        add
        nxb     (y),loop
;
        pop     #P{4,4}          ; get result
        return
        end
```

Figure 9-3, continued

TOBS version 1 results:

| run | array size | # of inst. | cycles | ave. inst. | r-hit rate | w-hit rate |
|---|---|---|---|---|---|---|
| 1 | 20 | 87 | 1567 | 18.0 | 94.2% | 93.4% |
| 2 | 200 | 807 | 9953 | 12.3 | 97.9% | 99.0% |
| 3 | 1000 | 4007 | 50937 | 12.7 | 98.4% | 99.8% |

TOBS version 2 (multiply array by a scalar) results:

| run | array size | # of inst. | cycles | ave. inst. | r-hit rate | w-hit rate |
|---|---|---|---|---|---|---|
| 1 | 20 | 87 | 1341 | 15.4 | 92.0% | 89.9% |
| 2 | 200 | 807 | 7748 | 9.6 | 97.9% | 98.2% |
| 3 | 1000 | 4007 | 38370 | 9.58 | 98.6% | 99.6% |

M68000 results:

| run | array size | cycles |
|---|---|---|
| 3 | 1000 | 129040 |

**Table 9-4:** Dot Product Test Results

```
;       matprod
;
; Large matrices product
; Matrices are 50X50.
;
matprod: domain
         interface      prod
;
        own
;size:  integer 2500     ; total size of mat1 & mat2
;rsize: integer 50       ; row size
res:    array[2500] of real     ; result
; Following is the actual parameters table
m1:     array[2500] of real
m2:     array[2500] of real
acp:    ACP     {<#ARF{RE},m1><#ARF{RE},m2><#ARF{RE},#I{2500}>
                                <#ARF{RE},#I{50}><#ARF{WRT},res>}
;
;
        global
        locate
matproda: edomain {}
        define
mprod:  element 0 of matproda
;
        code
prod: entry     DYNSIZE
        call    mprod,acp
        stop
        end
```

**Figure 9-4:** Version 1 Matrix Product Program
with remote procedure call

```
; Matrix product procedure
;
matproda: domain
        interface     mprod
; Following are the formal parameters.
mat1:   array[2500] of real {{0x7fff}{0}}
mat2:   array[2500] of real {{0x7fff}{0}}
size:   integer                 ; total size
rsize:  integer                 ; row size
result: array[2500] of real
;
        global
        define              ; define selectors
x:      element of mat1 mode step
y:      element of mat2 mode step
z:      element of result mode step
        own
zero:   real    0
        dynamic
row:    integer
col:    integer
;
        code
mprod:  entry   DYNSIZE
        aclr    row
        aclr    col
        set     z,#I{0},#I{1},#P{5,3} ; set selector for result.
loop1:  set     x,row,#P{5,4},#P{5,3} ; set mat1 row selector.
        set     y,col,#I{1},#P{5,3}   ; set mat2 column selector.
        push    zero
;
; Compute res[row,col]
loop2:  push    x
        mul     y
        add
        rep     (x),loop2
;
        pop     z               ; store result and step z
        ainc    row
        acmp    row,#P{5,4}
        ba      LT,loop1
;
        aclr    row
        aadd    #P{5,4},col
        rep     (z),loop1
        return
        end
```

Figure 9-4, concluded

```
;       matprod
;
; Large matrices product
; Matrices are 50X50.
;
matprod1: domain
         interface      prod
; Following are the formal parameters.
mat1:   array[2500] of real {{0x7fff}{0}}
mat2:   array[2500] of real {{0x7fff}{0}}
size:   integer                 ; total size
rsize:  integer                 ; row size
result: array[2500] of real
;
        own
res:    array[2500] of real     ; result
; Following is the actual parameters table
m1:     array[2500] of real
m2:     array[2500] of real
zero:   real    0
acp:    ACP     {<#ARF{RE},m1><#ARF{RE},m2><#ARF{RE},#I{2500}>
                                <#ARF{RE},#I{50}><#ARF{WRT},res>}
;
;
        global
        define                  ; define selectors
x:      element of mat1 mode step
y:      element of mat2 mode step
z:      element of result mode step
;
        dynamic
row:    integer
col:    integer
;
        code
prod: entry     DYNSIZE
        call    mprod,acp
        stop
```

**Figure 9-5:** Version 2 Matrix Product Program with local procedure call

```
;
; Local matrix product procedure
;
mprod:  entry   DYNSIZE
        aclr    row
        aclr    col
        set     z,#I{0},#I{1},#P{5,3} ; set selector for result.
loop1:  set     x,row,#P{5,4},#P{5,3} ; set selector to row of mat1
        set     y,col,#I{1},#P{5,3} ;set selector to column of mat2
        push    zero
;
; Compute res[row,col]
loop2:  push    x
        mul     y
        add
        rep     (x),loop2
;
        pop     z               ; store result and step z
        ainc    row
        acmp    row,#P{5,4}
        ba      LT,loop1
;
        aclr    row
        aadd    #P{5,4},col
        rep     (z),loop1

        return
        end
```

Figure 9-5, concluded

TOBS with remote procedure call:

```
tobs> r matprod
clear time and cache? (y/n) y
termination time: 6778220 cycles
Instruction executed:   517656
Average instruction execution time:     13.094067

Number of memory token references:
        cache: 62082970     other: 1877285      total: 63960255

operand cache statistics:
# of reads 4600314      # read hits 4598726      %= 99.965477
# of writes 896576      # write hits 895937      %= 99.928726
# of page faults = 72   %= 0.004658
```

TOBS with local procedure call:

```
tobs> r matprod1
clear time and cache? (y/n) y
termination time: 6742257 cycles
Instruction executed:   517656
Average instruction execution time:     13.024590

Number of memory token references:
        cache: 62081855     other: 1875849      total: 63957704


operand cache statistics:
# of reads 4605306      # read hits 4603823      %= 99.967796
# of writes 896545      # write hits 895907      %= 99.928841
# of page faults = 63   %= 0.004076
```

**Table 9-5:** Matrix product test results

Run #1: TOBS results with a remote procedure call.
Rub #2: TOBS results with a local procedure call.
Run #3: Motorola results.

| Version | # of inst. | cycles | ave. inst. | r-hit rate | w-hit rate |
|---|---|---|---|---|---|
| 1 | 517656 | 6778220 | 13.09 | 99.97 | 99.93 |
| 2 | 517656 | 6742257 | 13.02 | 99.97 | 99.93 |
| 3 | | 21664800 | | | |

**Table 9-6:** Matrix Product Test Results

### 9.3.5 Evaluation

The TOBS results show a large time overhead caused by the initiation phase, in which nonresident pages are brought in, the cache is filled and NNT pointers are resolved. A large part of the time overhead is created by the remote procedure calls which appear early in the execution of all three programs. A large remote procedure call overhead is common among object oriented systems (see [Wulf 81]) because of the necessity to create a new execution domain. Yet in TOBS we make it possible to avoid most of that overhead by using local procedure calls most of the time, as exemplified in the quick sort example. That makes the remote procedure call overhead almost fixed, and when the number of iterations increases, its effect on the performance almost diminishes.

We also found that references through selectors are costly in execution time. That is expected, since selectors are a kind of indirection. It is also due to the relatively large size of a selector (64 tokens) which implies several memory accesses. The use of partially resolved selectors (i.e. the object

pointer OBP, when it is valid) helped in reducing this effect. The cache memory is also very helpful in this respect, since selectors and table entries are used frequently and are held in cache most of the time. This is an area for further study of alternative implementations of selectors and recognition of special-cases for efficient processing.

The difference between remote and local parameter references was found to be minor. This was found for structured parameters, when referenced through selectors, yet the same result is expected with scalar parameters, because the address in that case is resolved once and does not change during the execution of the procedure.

As can be seen from the results, the TOBS design is competitive in execution speed with the Motorola 68000 assuming a similar technology of implementation. It should be noted that TOBS employs several mechanisms for reducing the average memory access time. This includes the Access Processor that fetches instructions and operands in advance, whenever there are no dependencies on previous results. This is supported by a look-ahead memory for instruction fetch and a look-aside memory for operand fetch. We also assume a wide data path (64 bits) to memory, that may be achieved by memory interleaving. The result is that the Execute Processor rarely has to wait for the memory.

# Chapter 10

# Conclusions

This research has carried the design of object-oriented architectures forward a generation by exploiting locality properties and decoupled access/execute processing in an attempt to overcome the known performance bottleneck in previous object-oriented architectures. The resulting processor architecture is implementable with a complexity similar to conventional micro-processors.

Performance evaluation of the design shows marked decreases in the overhead experienced by previous object-oriented architectures and also provides a basis for further optimization of the design. Comparison to a standard micro-processor, the Motorola 68000, shows that an implementation of TOBS in a comparable technology will yield performance comparable to the 68000. This is somewhat surprising, since the Motorola 68000 has undergone many years of optimization while TOBS is still in its first design cycle.

The conclusion which can be drawn is that the benefits of object-oriented architectures can be obtained at no substantial cost in implementation or performance. Additional optimization of object-based architectures should lead to even more favorable performance comparisons with conventional architectures.

There is much additional research to be accomplished. The performance evaluations should be extended over a wider range of programs to be sure no significant performance limiting execution patterns have been overlooked. There is a spectrum of design optimizations to be evaluated. The entire area of fault tolerance of object-based systems should be investigated. TOBS shows promise for fault localization and this property should be quantitatively evaluated by simulation.

The process of mapping from higher level languages to architectures offers rich opportunities for performance enhancement through exploitation of early binding.

## References

[Almes 80] G.T. Almes.
*Garbage Collection in an Object-Oriented System..*
PhD thesis, CMU, June, 1980.

[Browne 82] J.C. Browne and S. Todd.
An Object Oriented Capability Based Architecture.
In T.L. Kunii (editor), *Proceedings of the 16th IBM Computer Science Symposium.* IBM, October, 1982.

[Browne 83] J.C. Browne.
An Object-Oriented Formulation for Distributed Systems: Integrity Management.
1983.
To be published.

[Browne 84] J.C. Browne et al.
Zeus: An Object-Oriented Distributed Operating System.
In *Proceedings of the 1984 ACM National Conference.* ACM, 1984.

[Browne 85] J.C. Browne et al.
A Performance Model of a Fault Tolerant Distributed System for Evaluating Reliability Mechanisms.
In *Proceedings of the 1985 International Conference on Management and Performance Evaluation of Computer Systems*, pages 363-378. 1985.

[Dafni 85] G.J. Dafni.
*Generating Unique IDs in a Distributed System.*
Technical Report, Department of Computer Sciences, University of Texas at Austin, 1985.
Technical Report TR-85-05.

[Ekanadham 79] K. Ekanadham and A.J. Bernstein.
Conditional Capabilities.
*IEEE Trans. on Software Engineering* SE-5(5):458-464, Sept., 1979.

[England 74] D.M. England.
Capability Concept Mechanism and Structure in System 250.
In *International Workshop on Protection in Operating Systems*, pages 63-82. IRIA, Rocquencourt, France, August, 1974.

[Fabry 68] R.S. Fabry.
Preliminary Description of a Supervisor for a Machine Oriented Around Capabilities.
In *ICR Quarterly Report*, pages B1-B97. University of Chicago, Aug., 1968.

[Fabry 74] R.S. Fabry.
Capability -Based Addressing.
*CACM* 17(7):403-411, July, 1974.

[Fagin 79] R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong.
Extendible Hashing - A Fast Access Method for Dynamic Files.
*ACM Trans. on Database Systems* 4(3):315-344, Sept., 1979.

[Feustel 73] E.A. Feustel.
On The Advantages of Tagged Architecture.
*IEEE Transactions on Computers* C-22(7):644-656, July, 1973.

[Gehringer 79] E.F. Gehringer.
Variable- Length Capabilities as a Solution to the Small-Object Problem.
In *Proc. Seventh Symp. on Operating Systems Principles*, pages 131-142. ACM, 1979.

[Giloi 83] W.K. Giloi et al.
Object Addressing Architectures.
In *The Int. Workshop on Computer Systems Organization*, pages 202-210. IEEE, March, 1983.
conference should be added.

[Gottlieb 83] A. Gottlieb et al.
The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer.
*IEEE Trans. on Computers* C-32(2):175-189, Feb., 1983.

[Gottlieb1 83] A. Gottlieb, B.D. Lubachevsky and L. Rudolph.
Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors.
*ACM Trans. on Prog. Lan. and Sys.* 5(2):164-189, April, 1983.

[Graham 72] G.S. Graham and P.J. Denning.
Protection - Principles and Practice.
In *Proc. of the Spring Joint Computer Conference*, pages 417-429. AFIPS, AFIPS Press, Montvale, N.J., 1972.

[Hansen 82] P.M. Hansen et al.
A Performance Evaluation of The Intel iAPX-432.
*Computer Architecture News* 10(4):17-26, June, 1982.

[Hemenway 81] J. Hemenway and R. Grappel.
Intel's iAPX 'Micromainframe'.
*Micro Systems* :73-89, May, 1981.

[HEP 82] Denelcor, Inc.
*HEP Concepts and Facilities.*
Technical Report, Denelcor, Inc., Feb., 1982.

[Houdek 81] M.E. Houdek,F.G. Soltis and R.L. Hoffman.
IBM System/38 support for capability-based addressing.
In *proceedings of the Eighth Annual Symposium on Computer Architecture*, pages 341-348. ACM/SIGARCH, May, 1981.

[Jones 79] A.K. Jones et al.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
*ACM Operating Systems Review* 13(5):117-127, December, 1979.

[Jones 80] A.K. Jones.
Capability Architecture Revisited.
*Operating Systems Review* 14(3):33-35, July, 1980.

[Lamport 78] L. Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
*CACM* 21(7):558-565, July, 1978.

[Lanciaux 78] D. Lanciaux, L. Schiller and W.A. Wulf.
*Supporting Small Objects in a Capability System.*
Technical Report CMU-CS-78-107, CMU, Feb., 1978.

[Myers 80] G.J. Myers and B.R.S. Buckingham.
A Hardware Implementation of Capability- Based Addressing.
*Operating Systems Review* 25(13), Oct., 1980.

[Pleszkum 83] A.R. Pleszkum and E.S. Davidson.
Structured Memory Access Architecture.
In *Proc. of the Int. Conf. on Parallel Processing*, pages 461-471. IEEE, Aug., 1983.

[Redell 74] D.D. Redell and R.S. Fabry.
Selective Revocation of Capabilities.
In *International Workshop on Protection in Operating Systems*, pages 197-209. IRIA, Rocquencourt, France., August, 1974.

[Ricart 81] G. Ricart and A.K. Agrawala.
An Optimal Algorithm for Mutual Exclusion in Computer Networks.
*CACM* 24(1):9-17, Jan., 1981.

[Richardson 84] J.P. Richardson.
*Synchronizing Concurrent Access to Shared Data.*
PhD thesis, Berkeley, April, 1984.
Honeywell technical report CSC-84-19.

[Smith 82] J.E. Smith.
Decoupled Access/Execute Computer Architecture.
In *Proceedings of the 9th Annual Symp. on Computer Architecture*, pages 112-119. IEEE, April, 1982.

[Taylor 83] R.N. Taylor.
A General-Purpose Algorithm for Analyzing Concurrent Programs.
*CACM* 26(5):362-376, May, 1983.

[Wulf 74] W.A.Wulf et al.
HYDRA: The Kernel of a Multiprocessor Operating System.
*CACM* 17(6), June, 1974.

[Wulf 81] W.A. Wulf, R. Levin and S.P. Harbison.
*Hydra/C.mmp: An Experimental Computer System.*
McGraw-Hill, 1981.

# VITA

Gad Josef Dafni was born in Israel, April 18, 1942. He served in the Israel Defence Forces 1959 - 1962. Earned his B.Sc. degree in Mathematics and Physics from the Hebrew University of Jerusalem in 1967. From 1968 to 1969, he worked as a programmer in the Israel Finance Ministry, Jerusalem, Israel, in data processing applications. From 1969 to 1981 he worked in the Armament Development Authority, Israel. He earned his M.Sc. degree in Computer Science, with distinction, from the Hebrew University of Jerusalem in 1975. From 1981 till now, he is working on his Ph.D. in Computer Science at the University of Texas at Austin, under the supervision of Prof. J.C. Browne.
Gad J. Dafni is a member of the ACM and Phi Kappa Phi honor society. He is married and has one child.

Permanent Address: 1111 Rutland Dr. #71
Austin, Texas 78758

This dissertation was typed by the author.