CMU-CS-82-124

(12)

Execution Environments in Programming Languages and Operating Systems

Robert W. Schwanke

May 1982

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-82-124

Execution Environments in Programming Languages and Operating Systems

Robert W. Schwanke

May 1982

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

Execution Environments in Programming Languages and Operating Systems

Robert W. Schwanke

Abstract

Multi-tasking operating system design is a thorough test of a programming methodology. Such systems contain large and complex data and control structures, manipulate unsafe hardware, require very efficient code, and must execute continuously for days at a time in the presence of transient hardware errors. Furthermore, they must conform to real-time constraints of hardware and users, and still satisfy throughput requirements.

The module construct in most recent methodology-based languages specifies only the source language structure of programs. However, the structure of the executable representation of an operating system program is very complex, and need not be isomorphic to the source structure. The operating system designer needs control over the executable representation of the system, especially when programming bootstrapping facilities, system generation and configuration programs, interfaces to hardware dependent modules, and managers for such execution facilities as address translation tables, process state registers, interrupt vectors, dynamic storage, protection domains.

The goal of the thesis is to determine whether an explicit notation and methodology for describing the executable representation of a system of programs, can improve our ability to design and construct operating systems. We investigate this question by extending a specific methodology, *incremental machine design*, with notations and techniques for structuring executable representations. We exercise the extended methodology by applying it to a set of realistic operating system design problems.

The thesis defines an execution environment to be an explicit set of resources for implementing programs. In each environment, the list of resources defines the interface between the implementation language and the underlying operating system facilities. That is, the translator implements program units using the set of resources supplied by the operating system.

The methods proposed to carry out this theme include: representing execution environments as explicit modules in source-language programs; binding source language program units to environments explicitly, in a way that allows both multi-environment modules and mulit-module environments; including host machine and boostrapping execution environments in operating system descriptions; and combining all program units into a single comprehensive system description, which takes the form of a program to create an operating system.

Applying the system of methods to a set of operating system description problems demonstrates that it clarifies many dependencies among operating components, provides a sound place for system generation and bootstrapping code in the overall system structure, and provides a basis for integrating operating system facilities directly into systems implementation languages.

Table of Contents

List of Figures

CHAPTER 1
INTRODUCTION

Multi-tasking operating system design is a thorough test of a programming methodology. These systems contain large and complex data and control structures, manipulate unsafe hardware, require very efficient code, and must execute continuously for days at a time in the presence of transient hardware errors. Furthermore, they must conform to real-time constraints of hardware and users, and still satisfy throughput requirements.

Much of the recent research in programming methodology and programming languages has focused on the *module* as a schema for structuring programs. Conceptually, a module is a group of program components that share information about certain design decisions, such as the format of a data base or the queueing structure used by a process scheduler. Several recent languages have provided explicit module constructs, which permit the system designer to isolate a set of related components such that changes to the shared design information affect only the code within the module, and such that the interactions between the encapsulated components and the rest of the system can be identified, characterized, and controlled. With such a tool, a system can be described, developed, and debugged, module by module, resulting in a cleaner design, and facilitating error detection and design changes.

The module construct in most recent methodology-based languages specifies only the *source language structure* of *programs*. In *operating systems*, however, the structure of the executable representation of the program is also complex, and need not be isomorphic to the source structure. The representation of a single source-language module may be spread across several execution environments. Programs that invoke operations provided by a system's virtual memory manager, for example, may affect the addressability of objects in ways that don't appear in the source program. Current languages do not provide any tools for specifying the structure of the executable representation of a system of programs. The operating system designer needs such tools for bootstrapping facilities, system generation and configuration programs, interfaces to hardware dependent modules, and system components which directly manipulate execution facilities, including address mapping *managers*, *process managers*, exception handling mechanisms, and synchronization facilities.

The goal of the thesis is to determine whether an explicit notation and methodology for describing the executable representation of a system of programs, can improve our ability to design and construct operating systems. We investigate this question by extending a specific methodology, *incremental machine design* [Habermann 78], with notations and techniques for structuring executable representations. We exercise the extended methodology by applying it to a set of realistic operating system design problems. Then, we evaluate the notation and methodology with respect to usefulness and generality, clarity and precision, implementability, and robustness.

1.1. What's in a Representation?

The representation of a program is often thought of as "whatever the compiler produces" when presented with the program. However, if we start from a more formal definition of "representation", we are led to include a more extensive set of objects than those produced directly by the compiler. The dependencies among these objects can be complex, and different from source program dependencies.

We can borrow the definition of representation from data abstraction methodology [Hoare 72], in which the representation function of an abstract object specifies a mapping from the values of certain concrete variables to the value of the abstract object. A concrete variable may be involved in the representation of one or several abstract objects. When the variables named in a representation function determine the representation of that one abstract object only, we tend to think of them as *being* the representation of that object. When one concrete variable is involved in the representation of several abstract objects, we often say that the variable is *used to represent* those objects. For example, a List Element data type might be implemented by a single vector of list elements, with a central type manager which handed out pointers to individual elements as they were needed. The central vector would be *used to represent* the elements.

Many of the same ideas apply to the representation of programs. In conventional compiled language systems (e.g. Pascal), the representation of a procedure is a specific body of machine-language instructions. The representation of a process (in, say, Modula), however, would involve a stack of activation records, the process's state vector, the representations of all of the procedures it invokes, and portions of the process scheduler. The procedure code is needed to give an interpretation to the program counter and other state variables. The process scheduler data structures are needed to embody scheduling properties, such as estimated completion time. Thus the procedures and the process scheduler would be *used to represent* the active process.

Now, let us consider the nature of operating systems programs. An ordinary sequential program has a well-defined beginning, middle, and end. Its execution can

be understood quite independently from the process which carries out that execution. However, a great many commercial application programs, and most operating systems components, are cyclic and non-terminating [Fion 77]. They make use of data structures which endure (theoretically) for years; there may be one or several "daemon processes" continuously executing code of the program; and, there may be portions of the operating system's data structures permanently reserved for the needs of the application or component. Therefore, taking the word *program* in this large-scale sense, the *representation of a program* may involve processes and their representations, large data structures on secondary storage, portions of operating system scheduling tables, virtual memory mapping tables, and other operating system data structures. Not every structure will necessarily have components *dedicated to* the individual program, but the representation function of the program may include all of them in its domain.

Many of the interactions between program components cannot be derived solely from their source-language descriptions. This is primarily because the organization of the executable representation of the program may be very different from the source-language organization. To discuss this difference, we will need to be a bit more formal about programs and their components. I will continue to use the term "program" in the broad, informal sense used above. To speak precisely about the organization of programs, I will need the term *module*, as it is used in discussions of data abstraction. A *source language module* is a syntactic unit of the source language, defining a naming region (scope), an interface to other modules, and a set of program components, which may be variables, procedures, constants, macros, inner modules, etc. [Schwanke 78]. A program, then, is simply a "main module".

Parnas has pointed out [Parnas 71] that the runtime structure of a program can (and usually should) be quite different from its source-language structure, because the runtime structure is based on the phases of processing, whereas the design structure should often be based on the structure of the data. Consequently, there need not (and should not) be a coherent machine-language object which is *the* executable representation of a module. For example, some procedures defined in a module may be "inline" procedures, whose machine code representation will appear at each call site, rather than just in one place in the module's executable representation. The converse also holds true -- the module may invoke an inline procedure, whose machine-language body properly belongs to the representation of another module. Furthermore, we have just seen that the data of one module (the process scheduler) may be used to represent parts of several other modules (programs declaring processes). Finally, modules can be composed into bigger modules and systems of modules (often called *subsystems*), which might be spread across a number of execution environments and multiple operating system layers. Therefore, all of the mapping tables, resource managers, and protection mechanisms which hold the subsystem together, must be accounted for in its representation.

Since there is not a one-to-one mapping from source to executable modules, we shall instead define the concept of an execution environment. Then we can talk about the grouping of machine-language objects into execution environments, and the relationships between source modules and those environments.

An *execution environment* is a coherent set of program execution facilities. It is a receptacle for programs and data, such that a given machine-language instruction sequence would have the same meaning in any procedure placed in the environment. An execution environment may be characterized by:

- The regions of memory which are *addressable* by procedures in the environment.
- The memory which is available for containing procedures and data. (Memory may be *addressable* without being *available*).
- The set of legal machine instructions.
- The "virtual machine instructions" (e. g. system functions, pr[illegible]ted procedures, or ordinary procedures) which may be invok[illegible] by procedures residing in the environment.
- The program support facilities (sometimes called run-time f[illegible]) available for synchronization, exception handling, message passing, bulk storage, measuring time, general I/O, *et cetera*.

Execution environments need not be disjoint. For instance, two environments might be identical except that not all of the memory addressable from one is addressable from the other. This might be true of the environment containing the implementation of a page fault mechanism, which would forbid the use of any addresses which might generate page faults. Another example is two environments which share a program support facility. Two user environments might rely on a common virtual memory manager, whose data structures intermingle information pertaining to the two environments.

A good example of multiple, overlapping environments is the VAX/VMS operating system. The lowest level of that system is the interrupt handling code, which coordinates a variety of device communication tasks via cooperating interrupt handlers. This level provides an environment for subsequent system layers, in which devices may be thought of as processes which send and receive messages. The process scheduler resides on top of the interrupt level, providing an environment in which system and user processes can cooperate through shared data, event flags, and inter-process interrupts. Subsequent layers of the operating system refine and constrain these facilities, and also add facilities for process-private memory, multiple memory protection rings, and for swapping both system and user memory. Thus we may view each layer of VMS as executing in a distinctly different execution environment, and in turn providing enhancements to that environment for use by subsequent layers.

1.2. Motivation

My personal motivation for this thesis comes principally from my work on the FAMOS system. Although the *design* of that system was methodologically elegant, the *implementation* was an unending source of frustration. It came at a time when data abstraction languages were just beginning to appear in the literature. We developed a paper language to use as a design notation, which we translated into Bliss-11 code. During the later phases of the project we increasingly found pieces of the Bliss-11 code, pertaining to environments, which had no adequate representation in the design language, yet which we believed were methodologically sound. These pieces turned out to contain a disproportionate share of the bugs, consuming vast amounts of time during system integration and test.

Relationships among execution environments lie at the heart of an operating system, permeating and often defining its structure. High level languages still do not provide a notation for relating source language programs to execution environments, nor for relating the executable representation objects collected in an environment to the programs which implement that environment. This notational lack prevents a system designer from writing a unified, formal description of the structure of the system. Without such a description, claims about formal properties of the design (e.g. hierarchy) can neither be verified nor disproved. The *ad hoc* notations (e.g. linker command files) that fill the notational gap in real development efforts, violate the principles of information hiding and locality. They must be rechecked for consistency with the source code after every non-trivial design change. For instance, design changes to environment management facilities can affect not only disparate portions of the operating system, but also the compiler, linker, and loaders. Automatic programming aids, including type checkers and debuggers, cannot adequately draw together information from different subsystems, leaving a significant amount of consistency checking to be done haphazardly, at system startup, where debugging tools (likewise limited by the notation) are at their most primitive.

Execution environment management has played an important role in operating system design research. Multi-tasking, virtual memory operating systems first appeared in the early sixties. By the late sixties, multiple processes and virtual memory began to appear *inside* operating systems. For example, the THE operating system [Dijkstra 68] had at its lowest level a process manager; the rest of the operating system was written as a collection of cooperating, sequential processes. Multics [Janson 76], Hydra [Wulf 74a], and FAMOS [Habermann 76] all used virtual memory mechanisms to isolate system components from one another. CAP [Needham 77], MUSIC [Loehr 77], and DAS [Goullon 78] have all followed suit.

Many of these research operating systems were claimed to be *hierarchical*: some relation between system components was found which formed a partial ordering. Multics had a hierarchy of protection rings, THE had a hierarchy of work delegation,

Hydra eschewed hierarchy [Wulf 74b] for the kernel approach, and FAMOS had a functional hierarchy [Parnas 74]. On closer examination, however, the hoopla over hierarchy never seemed to pay off. One often sensed that the ordering relation was fine-tuned to fit the system, rather than the system being designed to fit a methodologically sound relation. The FAMOS system, for example, was supposed to be partially ordered by the relation, "X calls *and depends on the results of calling* Y". However, we never found an adequate model for our loader or debugger within that framework. We also found difficulties with this model for the upper levels of the hierarchy, when trying to organize modules managing memory, addressing, and processes. In order to make them form a hierarchy, we had to divide each manager into several layers, to be interleaved with layers from other managers, forming a hierarchical dependency relation. The price we paid for hierarchy was a modular decomposition within each manager that was unnatural in other respects. The Multics system suffered the same difficulty; Philippe Janson has proposed a redesign of that system that solves some (but not all) of the problems [Janson 76].

Even though multiple execution environments have become so common in operating systems, data abstraction languages have so far provided little help in describing the bindings between programs and execution environments. In conventional language technology those bindings must be established instead by a linker. The compiler translates source programs into sets of control and data sections; a linker, guided by a command file, groups sections into memory segments, attaching various tags and labels to each section; then, a loader files the segments into the environment management tables.

Using a linker command file to represent parts of a system's structure has severe drawbacks:

- The ragged interface between source language programs and linker command files seriously interferes with the clarity of the system description.
- Inconsistencies between the programs and the linker command files cannot be detected automatically.
- None of the programming aids developed for the programming language apply adequately to the program/linker interface.

One place where these shortcomings are felt most acutely is in the environment descriptors themselves. In FAMOS, for instance, each segment descriptor appeared in the source code as an element of the *segment table* portion of an *address space descriptor*. However, in order to know the parameters of an address space descriptor, one had to have compiled all of the programs residing in that environment, and have written the linker command file. The debugging option on the compiler changed the size of code sections, as well as creating data sections containing symbol table information. Consequently, throughout the debugging phase we were rewriting linker command files and address space descriptor declarations

almost daily. Inconsistencies invariably arose between them, which were not discovered until system startup. As with most development projects, a bug at that point often necessitated several hours of delay for recompilation and relinking.

Because environment managers must manipulate objects produced by the language system, and because the language system is not included in conventional system descriptions, interactions between environment managers and the language system cannot be documented properly, and can thus become quite ill-structured. For example, one of the procedure linkages in Hydra/C.mmp permitted inter-page procedure calls where the target page was not addressable at the beginning of the call. The calling sequence specified both a page identifier (an index) and the address of the procedure within that page. Support for this linkage required cooperation among the compiler, the linker, the loaders, and the debugger, without any notations or support tools for programming the interfaces.

Several recent attempts have been made to create languages for systems implementation that support multiple environments, including Concurrent Pascal, Modula, and Gypsy. [Brinch Hansen 75, Wirth 80, Ambler 77]. Each of these languages define synchronization constructs and device communication facilities that severely constrain the class of operating systems for which the language is appropriate [Loehr 77]. In fact, the run-time support software for these languages (particularly Concurrent Pascal) resembles the kernel of an operating system.

Some of the objects used to represent an execution environment are not only used *implicitly*, by the instruction execution mechanism, but are also named *explicitly*, in source programs, as parameters to operating system function calls. This can lead to inconsistency between the state of the execution environment as assumed by the language system, and the state as it actually exists. For example, languages that provide a formal synchronization mechanism do not ordinarily provide a means by which one process can abort the execution of another, because there is no obvious way of communicating news of the death of one process to the other processes with which it was cooperating. A similar difficulty arises in memory management mechanisms. In FAMOS we had a memorable bug involving addressability. One of us came across a piece of code of the form,

```
if <logical variable> then
    <statement 1>
else begin
    <statement 1>;
    <statement 2>;
    <statement 3>
    end
```

Thinking himself clever, he "optimized" it to say,

```
<statement 1>;
if not <logical variable> then
    begin
    <statement 2>;
    <statement 3>
    end
```

However, the resulting program didn't work! It turned out that <statement 1> invoked an operation on the environment's address mapping table, which removed the segment containing <logical variable> from the virtual address space. Consequently, the "optimized" code gave an addressing error.

Since so much of the static structure and actual code of operating systems involves execution environment management, a methodology and notation for constructing and manipulating environments would benefit operating system design and development greatly. Specifically, it would:

- Permit a comprehensive system description to include environmental interactions among components, thereby forming a broader basis for formal analysis of system structure.
- Provide a framework for various programming aids, including consistency checkers, system integration tools, and debuggers.
- Bring the benefits of data abstraction to the task of programming the connections between source programs, their executable representations, execution environments, and the managers of those environments.

1.3. Scope of the Thesis

This thesis develops and evaluates a system of methods for designing and constructing conventional multi-tasking operating systems. It is an extension of an existing methodology, *incremental machine design* [Habermann 78], and is also based on the methods of data abstraction [Wulf 76].

I presume an implementation language that supports strongly typed data abstractions, checked during compilation. The ability to precisely control the interaction of system components, and the ability to compose program units in a well defined way into progressively more abstract entities, are essential to coherent design and development of large operating systems. Furthermore, I expect the compiler for the implementation language to produce machine code that is acceptably efficient for system software.

This thesis addresses the description problems of large, complex operating systems, such as commercial multi-tasking systems for conventional virtual memory architectures. The techniques will of course apply to smaller systems, but I will not be satisfied with toy solutions to toy problems.

In line with my intent to solve "real" problems, I presume a compilation-based program development system, rather than one which is interpreter-based. Furthermore, the methodology must apply to cross-compilation environments, although it is not limited to them. (I mean specifically to exclude systems like L*, where one develops a system by "growing it" out of a simple interpreter. These designs beg the question of how one designs and develops the interpreter and assures its correctness.)

One aspect of commercial systems I won't address is confidential source code. This phenomenon of the marketplace unduly complicates the task of customized configuration. Nonetheless, configuration methods that protect proprietary software may eventually come out of the present work.

1.4. Preview of Results

"Methodology" literally means, "a system of methods". A *set* of methods is a *system* when the methods work together to carry out an organizing principle. A methodology is valuable when the principle and methods combine to produce better *programs*.

The organizing principle behind the proposed methodology is the following:

> An operating system and its implementation language are integral parts of one another. The language system has no resources of its own. Instead, it implements system components out of resources supplied by the operating system itself. The language provides notations that both facilitate and discipline use of the resources. Each *execution environment* is a set of these resources, provided by some level of the system to support subsequent levels.

The methods proposed to carry out this theme include the following:

- Representing execution environments as explicit modules in source-language programs.
- Using both compile-time and run-time mechanisms to enforce the boundaries of environments.
- Designing or selecting an implementation language whose features harmonize with the facilities of the particular system being designed.
- Binding source language program units to environments explicitly, in a way that allows both multi-environment modules and multi-module environments.
- Including host machine and boostrapping execution environments in operating system descriptions.
- Combining all program units into a single comprehensive system description, that is a program to create an operating system.

Applying the system of methods to a set of operating system description problems, we shall see that it produces the following results:

- A better *interrupt synchronization method* than is available in current languages.
- A strongly typed characterization of bootstrapping, that preserves the modularity and hierarchy of the system design.
- A demonstrably hierarchical source language specification of a real operating system.

1.5. Goals

What constitutes a "good" methodology? Because this thesis does not contain any theorems or software by which to demonstrate the success of the methodology, the reader and the author must agree upon a set of criteria by which to judge it. The criteria I propose are:

- Utility
- Clarity
- Fitness
- Flexibility
- Implementability

1.5.1. Utility

Adopting a new methodology requires a substantial investment in software support tools and programmer training. Therefore, the methodology must be sufficiently useful to justify the cost. We shall consider the methodology useful if it makes contributions to a varied collection of important operating system description problems.

A description problem is the task of capturing a class of information about operating systems directly in the text of the programs. For the information to be considered adequately captured, it must be checkable for consistency with other parts of the system of programs, and must be subject to the same encapsulation and redundancy standards demanded of conventional program components.

The problem set, to be broad enough, must represent a variety of different stumbling blocks of operating system implementation, including both theoretical problems and problems known to cost time and manpower in real system development efforts.

Section 2.2 will introduce a set of problems by which to test the utility of the methodology.

1.5.2. Clarity

The methodology must lead to system descriptions in which *all* of the connections between a module and its neighbors are clearly represented.

Perlis *et al* [DeMillo 79] have argued that *verifiability* is more important than *verification* for producing reliable programs. A notation must of course have a

precise, axiomatizable meaning, but it must also be simple and clear, to help the authors and readers of programs understand them. Any new notations for representing execution information must have meanings that are simple and formalizable, in line with the embedding language. Furthermore, although multiple languages may be necessary, for describing different aspects of the the system, the total number of them must be kept small, and they must be sufficiently harmonious that both human and automatic readers may easily understand the relationships between parts written in different dialects.

1.5.3. Fitness

The methodology must provide notations that fit the problem domains of operating system description. It must provide, or provide the means to construct, notations for the objects, structures, and operations with which the system designer must deal, *in a style which corresponds to the way the designer thinks and talks about the problem domain.* Thus assembly language would not be a fit language for writing scheduling algorithms, nor would APL be a fit language for interrupt handling.

The main thrust of data abstraction languages has been to let the programmer for any particular domain construct the notations which fit his domain. However, they succeed only to the extent to which the class of abstractions the language supports fits the class of abstractions needed for the domain. For instance, very few languages provide *iteration* abstraction constructs, and the ones that do aren't yet fully mature. Any new notations introduced must be motivated by the problem domain, unsupportable in current languages, and general enough to find broad application.

1.5.4. Flexibility and Transparency

The methodology must facilitate system construction without materially constraining system design. That is, it must be *transparent* enough to give system designer full use of the hardware's capabilities, and flexible enough to assist the designer in a broad range of design methods. While not all system design practices are worth supporting, any system design style which can be defended should be possible within the methodology.

The term "transparency" has several definitions in the computer science literature. I define the *transparency* of a programming methodology, and of its notation, as the degree to which it makes the underlying machine's functionality available to the system designer. A methodology *should* encourage certain uses of the machine and and discourage or prevent others; however, the notation itself should only conceal functionality that is demonstrably undesirable. For example, even though the Bliss-11 language has no GOTO statement, its rich set of control flow structures

and highly optimizing compiler support nearly all of the desirable code sequences that an assembly language programmer might use. This definition of tranparency is due to Parnas and Siewiorek [Parnas 72a].

1.5.5. Implementability

The methodology must lead to programs whose performance is competitive with conventional operating systems. Also, the program development tools dictated by it must be sufficiently simple that they are easier to verify than conventional operating systems, and that their implementation costs don't outweigh their benefits.

1.6. Outline of the Thesis

This thesis contends that a methodology for operating system design should incorporate an explicit notation and specific methods for relating the source-language system design to the execution environments in which it will reside. I will substantiate this claim by extending a particular methodology, *incremental machine design*, with a notation and methods for describing execution environments, and then using the extended methodology to develop solutions to several system design and development problems.

In Chapter 2 we will examine *incremental machine design* as it was used in the development of the FAMOS system. In that system we will see various examples of important relationships between the source-language programs and the execution environments in which they run, especially relationships which affect modularity and hierarchy. From the FAMOS experience we will extract three significant problem areas to address with the extended methodology: interrupt synchronization, bootstrapping, and verifying hierarchy.

In Chapter 3 we will generalize from the FAMOS address spaces to a broad model of environments, and develop programming and system design techniques based on the model. By examining various ways in which the representations of abstract objects may overlap, we develop a model that supports many degrees of cooperation between environments, from mutual suspicion to complete trust. Viewing compilers and linkers as providing the concrete representations of programs, leads to an abstract data type model for the relationships between compilers, linkers, and environments, thus providing the "missing link" between virtual machine levels in system descriptions. Within that framework we then study the programming techniques needed to incorporate environment information in source programs, including new techniques for controlling the visibility of names, binding program components to environments, relating system generation programs to system descriptions, and initializing multi-layer systems.

Chapters 4, 5, and 6 apply the new methods to the problem areas identified in chapter 2, both to elaborate the methods and demonstrate their usefulness. In Chapter 7 we measure the extended methodology against the goals set for it in section 1.5. Then, in chapter 8, we relate the methodology to other work in the area, summarize the contributions of the thesis, and outline directions for future work.

CHAPTER 2
FAMOS: METHODS AND PROBLEMS

In this chapter we review the design of FAMOS. The FAMOS project produced a methodology of operating system design. By reviewing FAMOS we can formulate the principles used in that design, see how the principles appear in practice, and look for design and implementation problems that the methodology does *not* address. This analysis will provide the basis for extending the methodology with techniques and tools for dealing with execution environments in system designs, in chapter 3. The second half of this chapter formulates three diverse operating system design problems, arising from FAMOS but not unique to it, which will be solved in later chapters using the extended methodology:

- Interfacing to inconvenient hardware
- Verifying hierarchical relations
- System integration

FAMOS is a suitable foundation for this investigation because programming methodology was a central issue in its design, and because it used many small protection environments to achieve modularity *within* the operating system. I do not claim that the methods used in FAMOS are superior to those of other systems, but only that they are suitable for extension to cover problems of environments. I will from time to time cite examples from other systems, to show that the issues I am addressing are of general concern.

2.1. The Methodology of FAMOS

The FAMOS project was an experiment in modular design of operating systems. It tested the feasibility of designing a *Fam*ily of *O*perating *S*ystems in such a way as to minimize the redesign and recoding effort required to create a new member of the family. The family members might differ in underlying hardware configurations (anywhere from a single minicomputer to a large multiprocessor system) and in expected application (from batch to interactive to real-time). The material shared between family members could be actual shared code, or it could be a shared module *specification*, with different *implementations* to satisfy different performance requirements.

To achieve sharing among family members, the FAMOS designers employed two strategies:

- They identified a substantial set of *design decisions* that could be shared among all family members
- They used program design and implementation methods that facilitated *module* sharing among family members

Note carefully the distinction between the two strategies, and why both are necessary: being able to share code between family members is useless unless their designs are sufficiently similar that some modules are worth sharing. Conversely, unless two family members can actually share code, sharing design decisions will not likely lead to savings in the overall cost of the systems, and the designs will likely evolve in divergent directions.

The FAMOS methodology applies to more than just operating system families. The dimensions of variation among family members are the the same dimensions along which a single, real operating system is likely to change over time. An operating system ought to be able to adapt to changes in the underlying hardware and in user needs, without radical changes in the basic design.

Since the emphasis of this thesis is on methodology rather than system families, we will organize our review of FAMOS around its programming methods, rather than around the shared design decisions. Three terms characterize the methodology of FAMOS:

- Incremental machine design
- Modularity
- Hierarchy

Incremental machine design denotes building a system as a sequence of software layers, where each layer defines a virtual machine, on which subsequent layers can execute. The virtual machine features provided by a layer come from one or more *modules*, chosen and specified such that redesign of one module is unlikely to require reimplementation of other modules. In FAMOS two or more features at different levels sometimes share a design decision, such that changing that decision would require that they all be re-implemented. Those features would all reside in a single module. Nonetheless, the entire set of procedures in FAMOS is partially ordered (a *hierarchy*) according to *functional dependency*, defined as "X depends on Y *iff* X calls Y and depends on the results."

We shall discuss each of these concepts in some detail, then summarize the important aspects as they relate to environments.

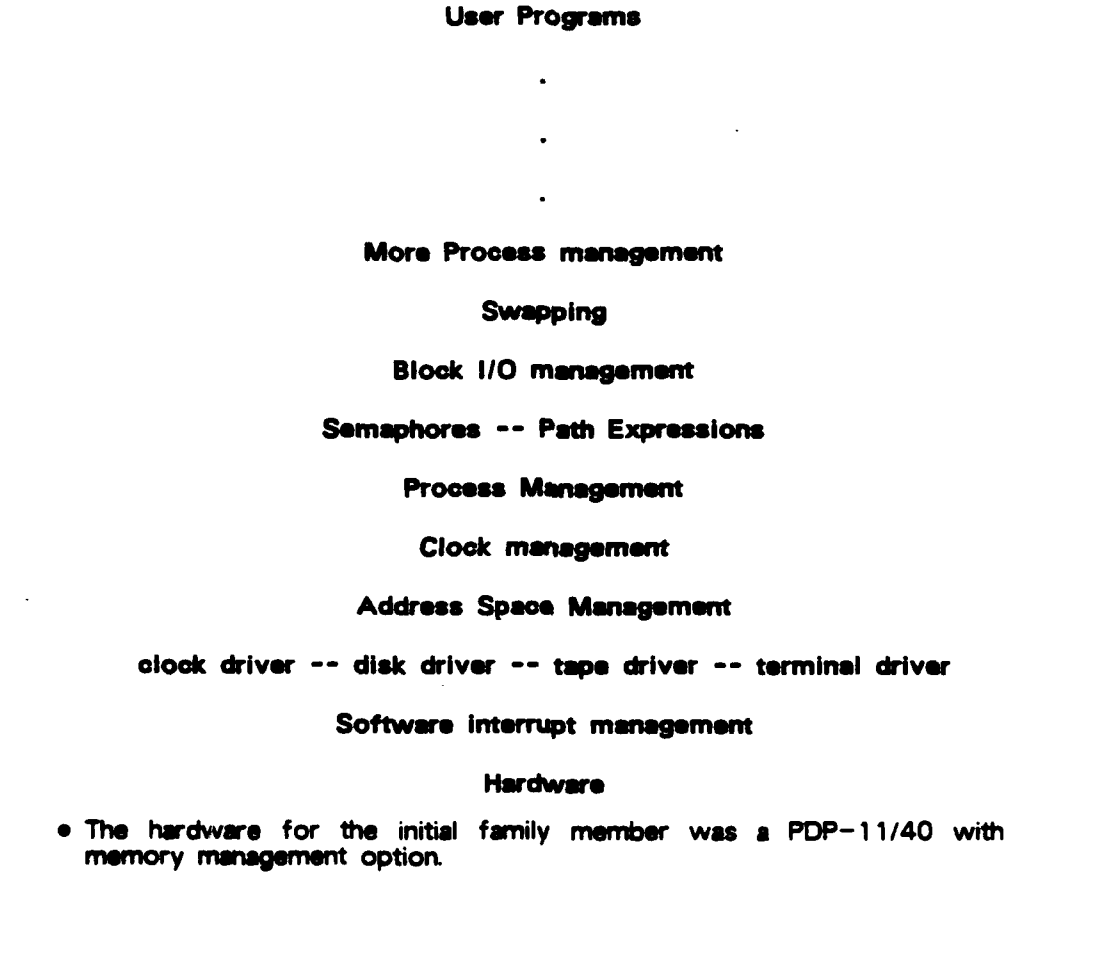
2.1.1. Incremental Machine Design

Incremental machine design has three important characteristics:

- Each layer enhances only a portion of the underlying system
- Transferring control from one layer to another does not necessarily imply any runtime overhead whatsoever
- The interfaces between layers can be viewed as virtual machines

Each software layer adds a facility or feature to the system, and conceals part of the underlying system. Specifically, each layer will only conceal those parts of the system that it uses; all other system features are freely available to subsequent layers. The resulting combination of new and old facilities defines the *virtual machine* on which subsequent layers execute. A layer may include several independent features, which are each built upon the underlying system, but do not make use of one another. For example, the two synchronization modules in FAMOS both use the process manager to maintain waiting sets, but are otherwise unrelated. The layers of FAMOS are shown in figure 2-1:

Figure 2-1: Software Layers in FAMOS



- The hardware for the initial family member was a PDP-11/40 with memory management option.

- The software interrupt queue provides a communication facility between interrupt routines and device management modules, and conceals the hardware priority register in the CPU.

- The device drivers conceal the physical device control registers, replacing them with "virtual devices" that are less time-sensitive, and that communicate with the device managers via protected procedure calls.

- The address space manager maintains static and dynamic address spaces, and implements protected procedure calls. It defines the state vector of a process, and provides context switching instructions. It conceals the relocation hardware, and all physical memory.

- The clock manager provides a set of interval timing clocks that can be be started and stopped individually. When an interval has elapsed, the clock manager invokes the appropriate wakeup routine via the software interrupt mechanism.

- The process manager keeps track of all "in-core" processes. It multiplexes the processor among those processes that are ready to run, and maintains the data type "waiting set" to hold sets of processes that have been blocked. The process manager uses and conceals one of the interval timers, to measure time slices and signal when each one ends.

- The semaphore facility and the path expression facility both use waiting sets to hold blocked processes, but are not directly related to each other.

- The Block I/O manager schedules transfer requests between primary and secondary storage, and blocks processes that are waiting for I/O. It conceals the virtual device registers that were provided by the corresponding drivers.

- The Swapping manager copies segments to and from disk, and permanently conceals a fixed portion of the disk, devoted to this purpose.

- The upper-level process manager initiates and terminates processes, and moves them between primary and secondary memory.

Because each software level conceals only those system features it uses, crossing a level need not cost any execution time at all. For example, the address space manager uses the software interrupt mechanism to report exceptional events, such as running out of free space. The memory manager may invoke the software interrupt mechanism as easily as the device driver can; no code is associated with "crossing" the device level to invoke the software interrupt level.

In FAMOS, each layer of the system is specified as a complete execution environment, with memories, processors, instruction sets, registers, and peripheral devices. At low levels, many of these environmental features are provided directly by the hardware; at higher levels each is implemented or in some fashion managed by the lower-level software.

For example, the clock driver executes in the privileged Kernel Address Space, is driven by ordinary hardware interrupts from the clock, and is able to manipulate the

clock control and data registers directly. Its execution environment includes only one programmed feature, the software interrupt queue. The clock driver implements an abstract clock that is functionally identical to the hardware clock, except that the abstract clock won't lose time if the clock manager is delayed in responding to an interrupt. Operations on the abstract clock are implemented as kernel calls (invoked by the EMT instruction).

The virtual clock manager, in contrast to the clock driver, executes in an unprivileged *user address space*, maintained by the address space manager. It communicates with the clock driver by invoking kernel calls, and by receiving virtual (i.e. programmed) interrupts from it.

The combined set of new and old facilities available at each level define a *virtual machine* in the sense that the available facilities are analogous to hardware features; more importantly, the level is completely specified, such that every correct and incorrect use of the virtual machine produces a well-defined effect, and no sequence of operations on it can break the underlying software.[1] Each virtual machine level defines a possible point of convergence between family members. For example, the virtual machine defined by the process multiplexor level could have different implementations for uni- and multi-processor configurations; two family members could be identical above that level, and different below it.

2.1.2. Modularity

The FAMOS system was decomposed into modules according to the criteria advocated by Parnas [Parnas 72b, Parnas 71]:

- Generality: A good decomposition should keep viable as many useful design alternatives as possible.
- Information hiding: Any system facility that could be implemented in several useful ways, should be concealed in a module whose specification does not reveal which implementation method was chosen.
- Sparse connections: The connections between modules are the assumptions the modules make about one another. A module's specification lists all of the information that may be assumed about it; in general this should be *much* less than the information that is known about it by its implementors.
- Hidden data structures: one aspect of a facility which is often redesigned is the organization of its data. Therefore, each data structure should ordinarily be concealed in a module that provides all of the operations necessary to access it, without revealing its organization.

---

[1] This statement assumes the implementation is type safe. In general, if a program bug at a higher level can produce an undetected address computation error (such as an array index out of bounds), the bug can propagate to any part of the system. The protected addressing environment facility of FAMOS is intended to provide firewalls for defense against such problems; we will discuss this more in the context of modularity.

- Design modules vs. execution modules: The operations which a module provides for its facilities may be implemented as procedures, macros, interprocess messages, or even access algorithms. With this flexibility, a single machine-language procedure could be made up of instructions derived from several different design modules, provided that the program development system includes an adequate translation tool.

The clock manager illustrates several of these principles. It is specified such that the clocks are not bound to any other system components, such as processes. That allows higher system levels great flexibility in how clocks are used, whether to measure time-of-day, CPU slices, experimental phenomena, or program performance. Conversely, the collection of clocks is specified as a *set*, rather than a vector or list, so that the data structure organizing the clocks can be changed later if the initial implementation is unsatisfactory. In fact, two implementations of the clock manager have been built:

- a linked list implementation, which handles an interrupt in constant time, but resets a clock in time proportional to the number of running clocks.
- a vector implementation, which handles an interrupt in time proportional to the number of existing clocks, but resets a clock in constant time.

Both implementations satisfy the same external specification; a particular family member would use the implementation best suited to its performance requirements.

The reader will have noticed that many of the principles of modularity listed above have become codified in abstract data type programming methodologies. The FAMOS developers employed data abstraction techniques in several ways:

- The "programming standard" for the project dictated that the individual program components be written as abstract data type managers.
- Each major system facility is specified as a manager of some type, where that type embodies some virtual machine feature.
- The protected address space facility was modelled after the *module* concept now found in many programming languages.

FAMOS is programmed in a strongly typed data abstraction language, for which no compiler exists. Instead the system is translated by hand into Bliss-11. The Bliss code reflects the type definitions and module declarations of the high-level description, although Bliss itself does not support typed variables.

The process manager is an example of providing virtual machine features as an instance of a type. It is specified as the manager for the types *virtual processor* and *waiting set*. Each virtual processor provides a complete execution environment identical to the virtual machine defined by the address space manager, except that a virtual processor does not provide context switching instructions. Instead, it provides the type *waiting set* with operations "block" and "wakeup". A waiting set is simply a collection of virtual processors that have been blocked pending some event. The "block" operation moves the invoking process form the ready set to the

specified waiting set. "Wakeup" selects a process from the specified waiting set and moves it back to the ready set.

An address space in FAMOS consists of a set of code and data segments, a set of entry points, and a list of "capabilities" representing the right to invoke other address spaces. A program executing in one address space invokes another address space by means of a *protected procedure call*, naming an entry point of the invoked address space, and passing parameters. An activation record for a protected procedure has its own address mapping table and execution stack. Thus there can be several processes executing in a single address space simultaneously.

The features of an address space are quite analogous to module facilities in modern languages: The entry points amount to exported procedure names; the capabilities are imported module names; and, procedures can share code and data freely *within* the address space, but not *between* address spaces. Furthermore, an address space can be very small or very large, so that conceivably each execution module of the system could be protected in a separate address space.

For example, in order to protect the process scheduling data structures from errors in other system components, the process manager's procedures and data are isolated in their own *virtual address space*. Like the clock manager's address space, it has no special privileges. The process manager simply uses and conceals the context switching instructions provided by the underlying virtual machine.

The process manager exemplifies close correspondence between a design module and its execution environment. All virtual processors descriptors, and all waiting lists, are stored in the process manager's address space. Each abstract operation is implemented as a very simple macro, that sets up and invokes a protected procedure call to the process address space. The correspondence is close because the type manager is highly suspicious of its users. Leaving any processor descriptor or waiting list unprotected would make the entire system vulnerable: damage to one of these could shut down the scheduler. However, there need not be a one-to-one correspondence between design modules and protection environments. Other modules in FAMOS are less suspicious of their users, and are correspondingly less protected:

- The *sorted list* module is used independently by several other modules. Each of them has a separate instance of the manager's code and data, placed in the using module's environment.
- The *semaphore* module uses a waiting set to hold blocked processes. The representation of the semaphore consists of a count field, placed in the declarer's environment for efficiency, plus a waiting set, kept in the process address space for safety. A bug which destroys the count field will certainly destroy the semaphore; however, it cannot damage the waiting set, nor the scheduler itself.
- The *clock manager* module represents a virtual clock with a clock descriptor, kept in the clock address space for quick access on

> interrupts, plus a programmed interrupt vector, kept in the address space manager's (kernel) address space. The clock manager has no control over access to the interrupt vectors. Neither does the address space manager check whether a request to set an interrupt vector comes from its owner. Consequently, the wakeup routine register of a virtual clock is not protected by any run time mechanism.

These examples serve to show that the partition of the FAMOS system into design modules is different from its partition into execution modules. Partition for design is based on concern for human comprehension; partition for execution is based on concern for protection, efficiency, space limitations, and other physical properties. The different partition for execution does not compromise the modularity of the *design*; it merely specifies mechanisms for run time checks on program integrity.

These examples also show different ways in which instances of a type appear in different execution environments. A methodology for dealing with environments should provide tools for programming each of these type management styles.

2.1.3. The Uses Hierarchy

Up to this point I have been somewhat informal about how the levels of the FAMOS system interact with one another. In fact, the interaction takes place via function invocation, i.e. by an operation of one level invoking an operation provided by another level. Furthermore, the interactions between system levels obey a *functional hierarchy, ordered by the uses relation.*

Hierarchy is a much-overworked term for a desirable property of system designs. It denotes a *partial ordering* of system components according to some relation. To define a hierarchy, one must state both how the system is divided into parts, and the precise nature of the ordering relation between parts. A well-chosen hierarchy, faithfully adhered to, can make a system much easier to understand, leading to better design, easier debugging, and more straightforward verification.

Parnas has surveyed [Parnas 74] the ordering relations most commonly appearing in operating system designs, such as *invokes, uses, gives work to, is composed of, gives resources to,* and *is more privileged than.* For example, the THE system consists of a set of processes partially ordered by the relation *gives work to* [Dijkstra 68]. Each process is responsible for servicing a queue of tasks, each of which it supposedly either carries out directly, or passes on to other processes. If each process in THE can be shown to either carry out or delegate every task given to it, then because of the partial ordering one can be convinced that all work will eventually be done. If the relation were not a partial ordering, but were cyclic, then tasks could be delegated indefinitely, and never carried out.

The FAMOS system is partially ordered by the *uses* relation, over the set of all operations defined by all virtual machine levels. The *uses* relation is defined as *X*

uses Y if and only if *X calls, and depends upon the results of calling, Y.* A program that calls procedure X ordinarily does not know or care whether or not X calls some procedure Y. Since a failure in Y would ordinarily make X appear to malfunction, X ordinarily depends on the results of calling Y. On the other hand, suppose X is a file manager, and Y is a file-user's "end-of-file" routine. In that case, X would be expected to call Y when the end of file was reached, and X would not be blamed if Y malfunctioned. Therefore, X would not be said to "use" Y, even though X would call Y.

Unfortunately, X *will* be blamed if Y enters an infinite loop, and never returns control to X. Therefore, to call a procedure without *using* it requires some guarantee that the procedure will terminate.

The FAMOS software trap mechanism embodies the "call-without-using" concept directly. It provides a "virtual trap vector" in which high level modules can insert protected procedure names. When a low-level module uncovers some condition that must be signalled (e.g. "*free space exhausted*", "*sleep interval terminated*"), it may invoke the protected procedure, without knowing what the procedure does, or depending on the outcome. It is the responsibility of the module defining the trap handler to certify its termination.[2] Roy Levin's exception mechanism [Levin 77] would include the FAMOS mechanism as a special case. We will explore the impact of exception protocols on modularity and hierarchy in chapter 4.

The *uses* relation, like the *gives work to* relation, aids proofs of system properties. In general, to verify that a module is consistent with its specifications, one must assume that the modules *used by* the given module meet *their* specifications. Consequently, if the *uses* relation is acyclic over a system, then the consistency of a system can, *one might conjecture*, be composed out of the consistency proofs of individual modules. This might not be easier than verifying the entire system at once, but the ordering should make verification easier regardless of the method, simply because the system is more regular than an unstructured one. Testing, too, *is simplified by a uses hierarchy*: any subsystem that is closed under the *uses* relation may be tested separately from the rest of the system. If the *uses* relation is a partial ordering, then the ordering defines a natural system integration sequence. By testing the lowest module first, then repeatedly adding the next lowest module and testing again, one can avoid having to debug a large number of interrelated modules simultaneously.

Although making the *uses* relation among functions be a partial ordering seems essential to good system design, other ordering relations affect system design,

---

[2] At one point during the system implementation effort, Cooprider proposed a change to the software trap mechanism that would protect the invoker from non-termination. There was not time to explore the feasibility of the proposal.

sometimes conflicting with the *uses* relation. For example, modern programming languages, and module interconnection languages, tend to highlight the *composition* relation and the *scope* relation, by means of a *module* construct. A module is said to be *composed of* the modules declared within it; it may also control the scope of the names defined within it, by making them available to, or hidden from, the surrounding context. The nested modules form a directed, tree-structured graph for both relations.

The relations *uses*, *is composed of*, and *controls the scope of*, interact in FAMOS as follows: A FAMOS module *controls the scope of* the names defined withing it. One function may *use* another only if the name of the latter is visible in the body of the former. The *uses* relation defines a partial ordering of all functions in the system; that partial ordering is partitioned into virtual machine levels, where the functions within a level *use* only functions defined at the same and lower levels. A virtual machine *is composed of* the functions of that level plus the next lower virtual machine. Figure 2-2 depicts these concepts: single letters denote functions, arrows depict the *uses* relation, the dashed box denotes a module, and the solid boxes denote virtual machines. Module M has chosen to conceal function C, so that only functions J and K may use it. The latter two functions are not concealed.

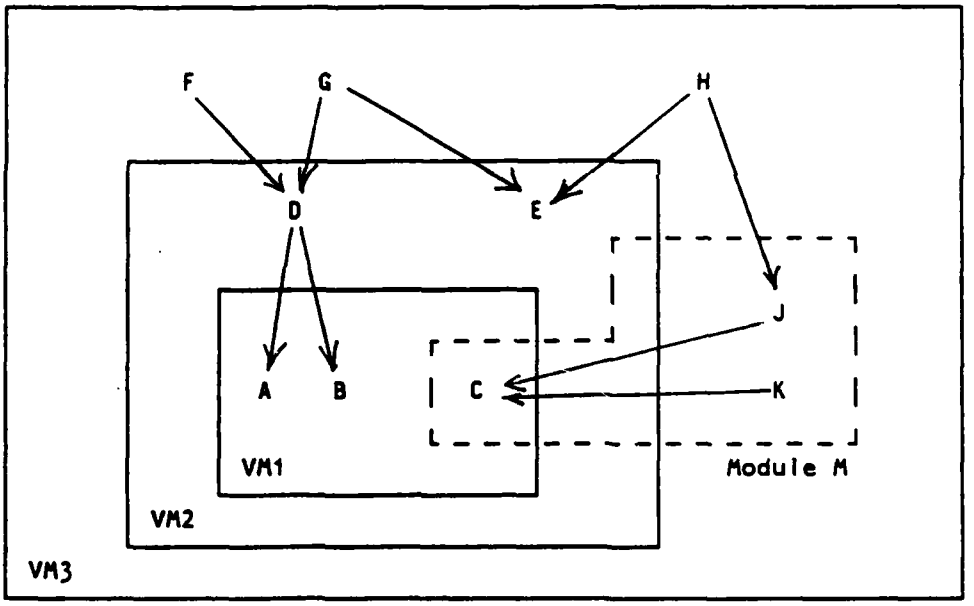


Figure 2-2: Functions, Modules, and Levels in FAMOS

Because FAMOS is organized as a hierarchy of functions, rather than modules, a single module could contain functions from several non-adjacent system levels. For example, the memory management module (cf. figure 2-1) implements the software interrupt layer, the address space management layer, and the swapping layer. This formulation of the module concept conflicts with recent programming language modules, because it makes the scope relation quite different from the composition

relation. For example, the virtual machine defined by the process manager level is composed of, among other things, the address space manager, and yet some of the features of the address space manager are concealed from the process manager, but available to the swapping manager. I will address this issue in section 3.2, presenting a new method for handling it, and apply that method in several of the case studies in subsequent chapters.

2.1.4. Summary

The design of FAMOS is organized as a sequence of virtual machines, from the hardware to the user environment. Each virtual machine is specified as an execution environment on which subsequent layers can run. A layer conceals only those underlying system features that it uses, leaving the rest freely available to subsequent layers.

FAMOS is programmed using abstract data type methods, despite the fact that its implementation language does not support typed data. It contains many examples where the partition of the system into *design* modules is quite different from its partition into *execution* modules. In particular it has many cases where several different execution environments contain instances of the same type.

Since the development of FAMOS, programming methods based on abstract data types have crystallized into languages such as CLU, Modula, Mesa, Euclid, Alphard and Ada. Several attempts have been made to build operating systems in data abstraction languages. In each case, the entire system executes in a homogeneous environment. Those system components that must reside outside that environment are either added to the programming language's "run time system", or simply programmed in some other language. In contrast to such systems, FAMOS is a multi-layer, multi-environment system. A strongly typed implementation language for such a system would have to include facilities for describing execution environment features, and for binding program elements to environments.

FAMOS *claims* to exhibit a hierarchical structure, based on the *uses* relation over the set of virtual machine operations. However, this claim is hard to verify, or disprove, because the bindings between virtual machine levels cannot be adequately expressed in available notations.

In order to determine with any confidence that a programmed system exhibits certain global properties, one must first be able to integrate all of the components into a single, comprehensive system description, where all of the connections between the pieces are represented explicitly. Without notations for representing environments, such a system description is far from complete, and the system designer must make do with *ad hoc* , piecemeal descriptions.

In the next section we will discuss three problem areas where the lack of an adequate methodology for dealing with environments, is keenly felt.

2.2. System Description Problems

In Chapter 1 I proposed to develop a methodology for dealing with execution environments, and to test that methodology by applying it to a set of operating system design problems. In this section I will describe three problems to use for the test:

- System integration
- Interfacing to hardware
- Representing hierarchy

Each of the problems arises in FAMOS, but is not unique to that system. Each involves environment description in a slightly different way. System integration requires environment descriptions to automate the translation, integration, bootstrapping, and startup of a system. The hardware interfaces in an operating system *are* environment features; they must be represented in system descriptions in a way that conveys their asynchronous nature, so that they may be coordinated with the synchronization facilities of the implementation language. To impose a hierarchical structure upon a system, *its representation must exhibit all dependencies* between modules, including those between program units and their environments. Together, these three problems are broad and deep enough to test the clarity, flexibility, and fitness of the proposed notation, and the overall usefulness of the methodology.

For each problem, I will first give a concise statement of the particular design task to be executed, then discuss the problem area in general, and finally give some desired properties of a solution to the task. The solutions to the problems will be presented in chapters 5, 6 and 4, using the extended methodology of chapter 3.

2.2.1. Hardware Interfaces

The architecture of the computer upon which an operating system runs determines in large part the character of the system. However, language facilities for coping with hardware features have so far been quite primitive, because hardware features themselves do not mesh well with the kinds of abstractions most often found in programming languages. Language designers have had to choose between providing each hardware feature in its naked state, disjoint from other language features, or providing it cloaked in some elegant language feature, making certain uses of the hardware quite convenient, but others impossible.

Device communication hardware has received a great deal of attention in systems

implementation languages, and so it is the focus of the design task for this problem area:

> Integrate a modern architecture's interrupt mechanism into a system design and system implementation language, such that all reasonable uses of the hardware are available to the driver programs, and the resulting facilities are available via abstract synchronization mechanisms, to higher system levels.

Three typical hardware facilities that are poorly represented in implementation languages are address translation managers, device drivers, and context switching operations. For example, an address translation manager will usually contain some data structure that resides on secondary storage, with pieces swapped into primary memory as needed. This overlaying of data must be done behind the compiler's back, since the compiler assumes that all data mentioned in a program will be present in the address space as needed[3]. We discussed in section 1.2 a bug from FAMOS in which a source program change that looked to be simply an optimization, in fact changed the address translation tables, leading to an "illegal address" error. Because the compiler knew nothing about address translation, it could not help detect the error.

Peripheral devices pose two kinds of problems to the system designer and language designer: giving the device driver programs access to the hardware, and synchronizing interrupt routines with the rest of the operating system. Operations on device control registers are intrinsically hazardous to the entire system, at least for devices that can read and write primary memory directly, because an incorrect value placed in a data register could cause the device to overwrite the wrong part of memory. No amount of type checking could prevent such an error. However, systems must have that access, so recent languages (e.g. Euclid, Modula) allow a program to bind a variable to a specific memory location, thereby giving source-language access to device registers.

Device synchronization problems are more difficult. The asynchronous execution of an interrupt routine fits poorly, at best, into modern synchronization constructs. Masking interrupts by manipulating the interrupt priority level register of the CPU, is likewise clumsy and error prone. Concurrent Pascal, Modula, and Gypsy have each taken the approach of transforming the arrival of an interrupt into some more abstract event, which could then be handled by a conventional synchronization facility. Concurrent Pascal provides message channels and monitors, Modula has device processes, monitors, and signals, and Gypsy has a general message system.

To provide these abstractions, the language systems have had to incorporate fairly elaborate run-time support packages, consuming significant amounts of memory and CPU time. Furthermore, these facilities actually constrain the class of operating

---

[3] A few language systems do exist that support overlaid data, but not for systems implementation.

systems implementable in each language. Concurrent Pascal precluded virtual memory operating systems, by concealing the interrupt structure inside its message queues. Modula supports only non-preemptive scheduling of system processes, in order to reduce the synchronization overhead for its monitor-like *interface modules*. Gypsy provides a message system as its synchronization mechanism, thereby constraining the message system component of the operating systems it is used to implement.

Context switching operations are even more difficult to support in type-safe languages than device register operations, because they must directly manipulate the representations of programs. For example, a protected procedure call mechanism must have access to facilities normally concealed by the language system, often including access to its own representation. The resulting facility must be formally integrated with a language system that will make it useful for subsequent layers.

A good methodology for dealing with hardware interfaces should give the system designer two notations: a transparent one for writing the programs that will deal with the hardware directly, and another that conveys the abstract role of the hardware component in subsequent system layers. For example, it would supply a transparent notation for writing and synchronizing interrupt handlers, and an abstract notation for synchronizing device managers with system and user processes. Notations for each of these already exist; the methodology would contribute the "glue" to connect levels written in different notations.

2.2.2. System Integration

The integrate-and-test phase of system development has received very little attention in programming languages, despite the large fraction of the total system development time it often consumes. Many of the bugs that cause delays are due to lack of coordination between system integration, generation, configuration, initialization, down-loading, bootstrapping, and startup. This lack of coordination is due in large part to the lack of a comprehensive system description tying together all of the pieces.

A realistic problem in this area must be large enough to draw in several environments and several levels. I propose the following:

> Give a set of module specifications for a simple operating system, including the levels that provide static and dynamic storage management, virtual memory, and process multiplexing. The specifications must show how system integration, initialization, bootstrapping, and startup interact with one another in this system.

Programming languages for individual compilation units have been comparatively readable and meaningful since Fortran, and axiomatizable at least since Pascal. In contrast, languages for system integration remained quite primitive until the work of Deremer and Kron [DeRemer 75], and are still not commonplace [Tichy 80].

Instead, most system integration information has been described by linker command files, and *ad hoc* system generation programs. These notations are inappropriate for two reasons: they don't support the kinds of system environment concepts the programmer is trying to describe, and they don't mesh well with the programming language used to describe the individual modules. Specifically, the linker command languages do not typically provide any way to relate the object files to the source language program entities they represent, nor to relate them to the operating system programs that will manage the environment in which they are to reside. This type of shortcoming leads to curiosities like the B5000 swapping manager, which would occasionally swap out the space manager. (This made all in-swapping impossible.) The bug came about because the distinction between swappable and non-swappable code was not adequately supported.

Module interconnection languages were invented to address issues of name control and system structure, which linker command languages could not handle. Tichy's *Intercol* language, for example, supports visibility control with an Ada-like module specification syntax, augmented with facilities to handle multiple versions of system components. However, Tichy's language describes only the source-language structure of systems.

Module interconnection languages provide the *basis* for automating the integration of a multi-level, multi-environment system. However, they still need a way of representing execution environments, in order to automate the collection of environment management data. Objects to be managed include both explicitly declared ones, such as processes, and objects created by the compiler to represent programs, such as code and data segments. *Ad hoc* schemes for assembling this information tend to violate Parnas' information hiding principle, by requiring *too much* connection information to appear in specifications.

Initialization, bootstrapping, and startup require coordination of hundreds of small bookkeeping tasks, in multiple execution environments, for several different purposes, in a very hostile debugging environment. By "initialization" I mean inserting a meaningful value into every variable. By "bootstrapping" I mean installing and beginning to execute a complete system on a machine that initially has no program running and nothing meaningful in primary memory. By "startup" I mean connecting the system software being bootstrapped to the environment (hardware and permanent data) present at load time. (This may include primary memory diagnostics, reading in root directories off disks, or establishing communication with a network.)

Difficulties arise because all of these activities must go on simultaneously, and because several of them require coordination between different execution environments. For instance:

- Some startup routine may fail because it attempted to use a data structure that was not initialized yet.

- The act of moving a piece of data from secondary to primary memory, then executing it, is usually considered a type breach.
- The flow of control during bootstrapping is opposite to the usual system hierarchy. Each virtual machine must, when done with its own startup, transfer to a routine provided by its user.
- The loader must cooperate with the host-machine programs that write the data it reads. Again, this interface is usually poorly specified.
- System resources (memory and processes especially) must be allocated statically (during initialization), and also dynamically (on behalf of users).
- The bootstrapping and startup code itself must reside in special execution environments that can be dismantled, and the resources reused, once the system is running.

Most of the above problems of system integration, initialization, bootstrapping, and startup occur because existing notations do not allow the programmer to coordinate these activities adequately. System generation is normally handled by an *ad hoc* program. Loading is divided into a down-loading program on the host machine, and a chain of bootstrapping programs on the target machine. Initialization is split between the host and target machines, and on the target machine it is embedded in the same procedures that do bootstrapping and startup, with no distinction among them.

A useful methodology for environment management should lead to system design techniques that:

- *Transmit program representation objects (e.g. process descriptors)* to their managers automatically.
- Minimize the amount of relocation and linking that must wait until startup.
- Eliminate the necessity of type-breaching between virtual machine levels.
- Encapsulate decisions that affect code on both host and target machines
- Rationalize the startup of successive system layers
- Dispose of "startup code" cleanly

2.2.3. Representing Hierarchy

In section 2.1 we discussed the benefits of modularization and hierarchy in operating system design, but concluded that without explicit representation for dependencies involving environments, claims about hierarchy would be impossible to verify. To see whether the proposed methodology satisfactorily captures such dependencies, I will attempt to

Give a program decomposition for a system that supports

multiprogramming, space allocation, and paging, and swapping, such that the "uses", "composition", and "environment" relations are all apparent in the source code, and determine whether the relations are hierarchical.

Operating systems are promising candidates for the benefits of modularity and hierarchy, because they are large, complicated, long-lived, and continuously evolving. Furthermore, they must be able to recover from hardware errors gracefully, and detect software errors without destroying user data irrecoverably. However, finding a good hierarchical, modular system design is very difficult, because the components of an operating system interact with one another in subtle and complex ways, e.g.:

- Process management is usually separated from address space management, yet memory mapping information is part of the execution state of a process, and swapping managers need process synchronization facilities.
- Device interrupt routines often suspend the execution of the current process, then invoke synchronization operations provided by the process scheduler.
- Access to devices, interrupt vectors, and processor registers is often provided in the form of special memory locations, thus involving the memory manager in each of these other facilities.

Both FAMOS and MULTICS became entangled in a chicken-and-egg problem involving process managers and memory managers: a process manager *uses* its memory manager to move processes between primary and secondary memory; a memory manager *uses* its process manager to keep track of processes that have incurred page faults or need more memory. *The heart of the problem is that a process* needs both memory and a processor to run. Both FAMOS and MULTICS [Reed 76, Janson 76] had to divide up the relevant management programs in fairly unconventional ways to achieve hierarchical organization. The resulting programs had two or more versions of several descriptor types: a version for permanently existing descriptors, to be multiplexed by a low level manager, and a version for descriptors to be created and destroyed at will by a higher level module.

Modern data abstraction languages have contributed a great deal to modular, hierarchical system design. The relations "is composed of", "calls", and "has access to", can all be documented directly in the source programs, with the help of module definition facilities to prevent unintended dependencies. However, these languages come up short in supporting other relations:

- *Partial Composition*: In incremental machine design, each system layer conceals only a part of the underlying system. In current languages it is hard to say that one module is composed of "part of" another module.
- *Uses*: Current languages make it hard to distinguish procedure invocations that imply *uses* dependencies, from ones that do not. Current frontiers are exceptional condition handlers and iteration.
- *Scope*: Ideally a program component should only be able to *name* the facilities it is allowed to *use*. However, the scopes of identifiers in

most languages are tree-shaped, following the composition rules, whereas the *uses* relation is often a directed, acyclic graph.

- *Environment*: Bindings between source program entities and their environments, are not describable in modern languages. Consequently, many environment-based dependencies must be lumped under the *calls* or *uses* relation, without differentiation.

Data abstraction does indeed solve well the problems of data and program *composition*. It generally assumes a base environment of independent primitive objects, which can be composed into more abstract objects, and manipulated by abstract operations defined by procedures. This methodology does not apply equally well to data *decomposition*, however. In conventional language systems the representation of programs is left to the language system implementor; the programmer is allowed to think that he can create objects that are independent of each other. In an operating system, however, there is only one basic object: The Machine. The job of the operating system is to *decompose* the machine into independent objects that can be used to support independent program entities. For example, on most DEC systems all of the interrupt vectors must reside contiguously in physical memory, even though they have nothing directly to do with each other. The module that manages the interrupt vector table must arrange to decompose that table into independent interrupt vectors, each paired with a device control register, or some other hardware feature. Similarly, the very memory in which programs and data reside is tightly coupled, through whatever mapping mechanism is used to achieve virtual memory. The operating system must enforce a discipline on the use of that mapping mechanism that will preserve the appearance of independence of individual program entities.

The "chicken and egg" problem of FAMOS and MULTICS involved the relationship between static and dynamic versions of an object, environmental dependencies between memory and process management programs, and "uses" dependencies that did not follow the compositional hierarchy. An effective methodology for environment management must facilitate identifying *all* dependencies between program components, and recording those dependencies in the program text. It should also contribute some insight into the relevance of the various ordering relations that have been advocated for structuring systems.

2.3. Summary

The three system design tasks posed in this section form a diverse sampling of important problems involving the role of environments in system descriptions. The system integration problem has enormous practical implications, because it addresses an expensive aspect of system construction which has previously eluded formal treatment. The hardware problem is an interesting operating system topic by itself, because it must interact both with synchronization and exception-handling. It also

raises general, practical questions, such as how much effort should go into tailored language support for small system components. The hierarchy problem attempts to bring theory a big step closer to practice, by capturing the ordering relations directly in programs. In doing so we will be much better able to tell whether the proposed hierarchies actually lead to reliable, economical software.

The tasks were inspired by difficulties in the implementation of FAMOS, but are common to many operating systems. Their solutions should give us a measure both of how broadly useful the methodology is, and how deeply it penetrates three diverse areas.

CHAPTER 3
TECHNIQUES FOR ENVIRONMENT MANAGEMENT

The aim of the methodology is to describe an operating system completely within a strongly typed notation, including the bindings between programs, environments, environment managers, language support systems, and the physical components of the machine. Such a complete description will be a valuable aid to system design, integration, and verification.

The system of methods I present in this chapter synthesizes

- the *incremental machine design* techniques used in the Family of Operating Systems [Habermann 78],
- the *data abstraction* techniques represented in languages such as Alphard and Ada,
- software management techniques based on a module interconnection language [Tichy 80]
- a new conceptual framework and notation for environment management.

Our discussion of FAMOS in Chapter 2 gave an intuitive feel for the interaction of environments, modules, and shared data in that system. We begin this chapter by defining an environment as an *explicit* list of memories and instruction execution facilities. We discuss the features that may appear in environment specifications, and how they relate to operating systems and to language systems.

Incremental machine design methods require that a machine layer conceal only those parts of the underlying machine that it uses. Binding environment features to the modules that build upon them, requires a notation for the concept of *exclusive access*, which is different from simple access and from ownership by containment. We shall introduce such a notation, derive scope rules to support the concept of partial concealment, and show how the notation clarifies the dependency relationships among program modules.

Interfacing the implementation language system to a comprehensive operating system description requires an integrated approach to the design of the two systems. Because a systems implementation language should provide a fit, transparent notation for using the features of each virtual machine layer, I propose that the operating system *be* the run-time system for the language. To ensure harmonious

design, the system and its implementation language should be designed together. An environment specification defines the interface between the language and the system, expressed as a set of types, variables, and procedures provided by one and used by the other. For example, the operating system provides the types *section* and *segment*, which the compiler uses to implement types such as *procedure* and *variable*. More generally, a language would specify the minimum set of features every *environment module* must provide. This view of languages and environments lets us represent both compilers and environments as modules in comprehensive system descriptions.

Programming in a multi-environment context requires the ability to place instances of a single type in different environments. *Coupling*, through shared data, between a type manager and instances of the type, affects the ways in which the scattered type instances can be supported. We shall survey several different type management styles, seeing how coupling occurs in each, and how each can be used in multi-environment systems. We shall look at a specific case from FAMOS to illustrate the ideas.

Next we propose a language mechanism for binding program units to environments. The mechanism is designed according to the following principles:

- Binding a program element to an environment consumes resources, and therefore should normally be controlled by explicit program directives.
- A program unit may contain as much or as little binding information as desired, including none at all. Binding directives should not clutter programs unnecessarily.
- Binding a compound program unit to an environment should bind its components as well, but in a way that is flexible enough to support the full spectrum of type management techniques.

First I shall introduce a simplified notation for types and modules that clearly separates definition from instantiation, and add to it a syntax for environment annotations. I shall propose inheritance rules for propagating environment bindings to inner modules, and show that these rules harmonize with the type management styles identified earlier. By programming some examples derived from FAMOS, we shall see that the language mechanism satisfies the goals of brevity, modularity, information hiding, and utility.

To coordinate host machine and target machine activities in the system generation process, I propose to view the host machine as one of the environments in which the operating system resides. I shall program a small but realistic virtual machine, using the new notation, showing the relationships between compilers, linkers, loaders, initialization, bootstrapping, and startup. At the topmost level, the system description is a *program to create an operating system*.

3.1. Environments

In this section I shall define the concept of an environment. I take the view that an environment is a definite entity, corresponding somewhat to a virtual machine. A system designer creates environments, and places system components in environments.

An environment is neither a source-program *name context* nor a run-time *protection environment*. Instead, an environment is a source-program list of execution facilities. A module may make use of several different environments, and conversely, an environment may support several different modules. Furthermore, a single execution environment may span several protection environments, and a single protection environment may support several execution environments. An enviornment need not be protected at all, or its protection may be implemented by a combination of source language and run-time mechanisms.

Figure 3-1 shows how modules, environments, and protection facilities relate to one another, with respect to the *memory* portion of an execution environment. The system consists of two segments, S and T, two environments, E and F, and two modules, M and N. Environment E allows access to both segments S and T, whereas environment F only allows access to segment T. Module M uses only environment E, while Module N places procedures in both E and F. Procedure C can *name* the variable B because B is visible in Module N; C can also *access* B because they reside in the same environment. In contrast, Procedure A cannot *name* C because C is not visible in module M. Procedure D can *name* B, but cannot *access* it, because they reside in different environments, *even though they reside in the same segment.*

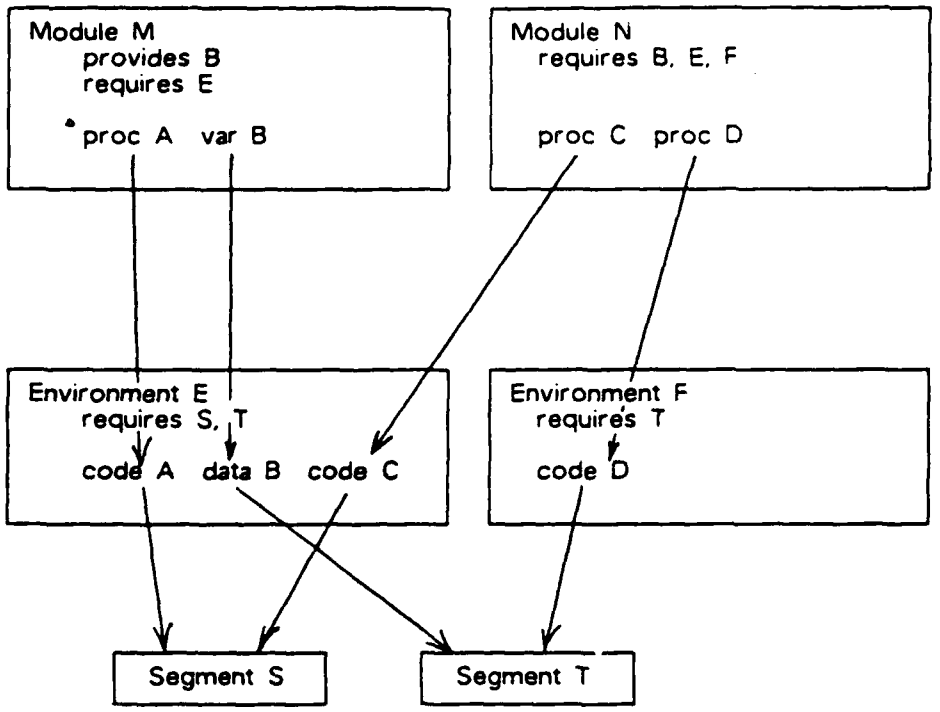


Figure 3-1: Modules, Environments, and Segments

The features of an execution environment can be roughly classified as *memories*, *instruction interpreters*, *registers and devices*, and *entry points*. We shall first discuss each class of feature as it appears in operating systems and language systems. Then we shall define in more detail the minimum set of features an environment must have to support the execution of programs.

3.1.1. Memories

A *memory* in an execution environment is any block of storage accessible by instructions executing in that environment. In operating systems we find both *virtual memory* and *physical memory*. We also find the notion of a *name space* (e.g. in Multics). We must be careful to distinguish among these concepts.

We define the *logical memory* of an environment to be the set of objects (variables and procedures) that a program executing in that environment can *access* (read, write, or call). For the purposes of this thesis we need to define a global naming system for objects, independent of individual execution environments. Each object in the system shall have a unique identifier, its *logical name*. Each logical name will denote a *logical address*, which specifies the memory location containing the object. Typically, objects are collected into segments, for administrative purposes. When this is the case, a logical address is composed of a segment name and a displacement within the segment.

Any given system may or may not provide a concrete realization of such a naming scheme. A good software development control system might have a global symbol table with logical names and addresses. On the other hand, a traditional compile-link-load system might only have segment-relative names, and a segment identifier might only be the name of a linker output file in the host file system. Specifically, however, not all objects having logical names will reside in the segment management facility of the system under construction, since some of them will be used to *implement* segment management!

A *virtual address* is the means by which an individual machine language instruction, within a given environment, refers to a particular location in the logical memory. The language system is responsible for constructing the correct virtual address for a given logical memory location. An operating system can move pieces of the logical memory from place to place in the physical memory, and can even retain several (identical) copies in different places. The execution environment is responsible for translating each *virtual address* into the current *physical address* of the designated location. However, the correspondence between virtual addresses and logical addresses need not be static, nor one-to-one. It is both conceivable and feasible in many systems to have more than one virtual address correspond to the same logical location. Also, in environments where the total size of the logical memory is larger than the virtual address space, such as in Hydra, FAMOS, and RSX-11, it becomes necessary to change the mapping between them dynamically, so that the "working set" is addressable.

Since a *logical address* is defined to be independent of any particular execution environment, I define a *local name*, or simply *name*, to be the environment-relative representation for a logical address. For example, a Hydra execution environment provides a "capability page set" (CPS) listing the segments (pages) that are accessible from that environment. A program refers to a logical page, in a system call, by its CPS index.[4]

To illustrate the relationship between local names, virtual addresses, logical addresses, and physical addresses, consider the following scenario from Hydra. A program in some environment wishes to increment the variable V. V is located in segment S, at displacement d. The logical address of V is <S,d>. The language system reserves a place, C, in the CPS for the segment S. The local name for V is <C,d>. When the time comes for the program to access V, the language system asks Hydra to make segment C addressable through relocation register R. The virtual address of V at that time is <R,d>. At the same time, Hydra places logical segment S in physical page frame P, making the physical address of V at that moment be <P,d>.

---

[4] Hydra actually uses the term "local name" to refer to elements of the "local name space", which is the set of capabilities a program running in the environment may use. I confine my use of the term to refer to objects that may be made addressable.

The Hydra addressing mechanism provides a separate CPS and relocation register set for each addressing environment. However, two environments within an operating system might share address translation facilities. For example, the device drivers and the address space manager in FAMOS coexist in the kernel address space, even though their logical memories are nearly disjoint. The fact that their logical memories were designed to be disjoint, except for one segment, is sufficient basis for saying that they reside in different environments.

Among the segments in the logical memory of an environment, we must distinguish between those that are merely *accessible* from the environment, and those that actually contain the *programs* residing in the environment. An environment might provide a set of utility procedures, in an execute-only segment. The programs compiled to execute in that environment would occupy storage allocated from a different segment. I define the term *program region* to denote the memory that contains the programs residing in an environment.

3.1.2. Instruction Interpreters

The basic instruction set for a given environment is simply the set of opcodes acceptable to the CPU's instruction interpreter. This set may vary depending on the privilege level in the program status register, but the possibilities are generally fixed once the underlying architecture has been constructed and the microcode written. However, from time to time we will want to designate certain procedures as "virtual instructions". For example, a "system call" instruction usually takes an immediate operand designating a particular privileged subroutine. Each such subroutine can be viewed as implementing a virtual instruction, whose "opcode" is defined by viewing the immediate operand as an extension to the basic opcode.

Protection-oriented operating systems often provide a "protected procedure call" instruction, via the above mechanism. This raises the possibility of viewing protected procedures as virtual instructions also, especially those procedures that perform utility services, and those procedures that support high level language features, such as synchronization and files. Finally, there are times when ordinary procedures are viewed as virtual instruction, such as when they are supplied by the language system to make up for omissions from the basic instruction set.

3.1.3. Registers and Devices

The assortment of registers and devices accessible in an execution environment would include physical devices, virtual resources, and components of the environment's instruction interpreter.

A physical device is often accessible only through virtual instructions, which

protect it from abuse and also make it more convenient to use. The clock provided by the FAMOS clock driver is such a device.

Virtual resources come in at least two varieties: those that embody physical resources, and those that simply provide system services. FAMOS virtual clocks illustrate the former; waiting sets illustrate the latter. In both cases the representations of the objects must be kept in a pool where the resource manager can access them at all times. In FAMOS they were also outside the logical memories of the environments that use them, although this would not always be necessary.

Instruction interpreter features might include a virtual address translation facility, a capability list, an exception handler, and a synchronization facility. These differ from ordinary devices and resources in that operations on them affect the instruction interpretation process itself. A successful methodology for environment management should coordinate the language system and the operating system with respect to such features. For example, the language system must know the correspondence between logical and virtual addresses at all times. It should also be able to relate the trap mechanism of the operating system to its exception handling and synchronization features.

3.1.4. Entry Points

An *entry point* is a designated logical address in the program region of an environment, to which control may be transferred from some other environment. This concept is intimately connected with virtual instructions and protection. Depending upon the protection mechanism separating two environments, a call from one to the other might be implemented as a macro, an ordinary jump-to-subroutine instruction, a trap instruction with literal argument, or a special protected procedure call instruction, with operands specifying the environment and address. The protocol for any procedure is part of its specification. In friendly systems, the entry point information could be the caller's responsibility, just as an ordinary assembly language procedure relies on its caller to use the correct starting address. In a protection-oriented system, an environment specification could include a dispatch table, to contain the addresses of legitimate entry points. The protected procedure call instruction would then take operands specifying an environment and an index into its dispatch table.

3.1.5. Definition of an Environment

The concept of environment I wish to define should satisfy the following properties:

- It should suggest how a system designer might *define* and *construct*

environments as concrete entities, rather than *deriving* them from other information. It must be possible for a system designer to select the number and nature of the environments in his system, name the memory and instructions in each, and name the runtime mechanisms that will implement them.

- It should reflect *only* the machine code aspects of an execution facility, and not source language constraints. In particular, it should not constrain the scope of identifiers.
- It should permit one to partition a program's executable representation independently from its source language organization. It must be possible both for an environment to contain components of several source modules, and for each of those modules to span several environments.
- It should support transfer of control between environments.
- It should give the system designer the flexibility to trade off between compile-time and run-time protection mechanisms.

Therefore, we shall define an *execution environment* to be

> a specification for a virtual machine, sufficiently detailed and complete that one could program that machine.

Such a specification would include, but not be limited to:

- a set of *logical memory* segments whose contents are accessible to programs residing in the environment
- a *program region* within the logical memory
- an *address mapping function* defining the relations between logical memory and virtual addresses
- an *instruction set*, including both basic and virtual instructions
- instruction interpreter *registers*, such as relocation facility and capability list
- an *entry point* mechanism
- other machine features

An environment specification could be represented in a strongly typed language as a *module specification*. This will let a system designer use source language facilities to define and construct environments. In section 3.3 we will discuss methods of doing this.

An environment constrains the use of identifiers only by defining the logical memory and address mapping function. An identifier in a source language procedure denotes a *logical address*, independent of any environment. Only when a procedure is bound to a particular environment must the language system translate the identifiers it uses into local names and virtual addresses. (In single-environment systems this is customarily done by the linker.) In section 3.4 we will discuss several type management styles, and assess the implications of placing type instances in several different environments.

More generally, environment specifications allow a language processor to determine whether a given source language unit is feasible in a specific environment. In section 3.5 I will define a syntax for binding individual program units to environments, and develop rules for propagating bindings to inner units, such that multi-environment modules are feasible. In particular, I will define a means of designating entry points to environments, and for analyzing the feasibility of passing parameters between environments.

The proposed approach to specifying environments not only distinguishes environments from source modules, but also distinguishes them from the run-time mechanisms used to implement them. This allows a system designer to use compile-time mechanisms for enforcing protection. For example, the page-fault handler in an operating system ordinarily cannot be allowed to generate a page fault. It must only access pages that are already in core. And yet, the handler code is ordinarily mapped by the same page table as the other time-critical procedures of the system. A system designer could define an "in core" environment, whose logical memory contained only segments guaranteed to be in core. This environment could use the same page table as other system environments, since the language system would insure that only safe pages were actually used.

To see how environment specifications would fit into a system description, in section 3.6 I shall develop a small example that shows the relationships between several environments in a small bootstrap loader. Chapters 4, 5, and 6 illustrate the same techniques in more realistic domains.

3.2. Partial Concealment of Machine Layers

Before pursuing the details of environment management, we shall discuss a shortcoming in existing facilities for scope control in high level languages, and propose a remedy. I will use the remedy to help describe dependencies between modules. This issue affects all operating system features, not just those that implement environments. However, since it does not directly affect the management of individual environments, the reader may want to skip this section on first reading. Section 3.3 begins on page 56.

Each of the virtual machine layers of FAMOS provides a rather long list of machine features for use by subsequent layers. However, each level is only *incrementally* different from the previous level. A module at a given level enhances just a few of the features provided by lower levels, leaving the rest available, unmodified, for use by other modules. For example, one machine level in FAMOS provides an address space management facility, a process multiplexor, a clock manager, and a variety of I/O devices. (Cf. figure 3-2.) One of the modules at the next level uses the address space manager and one of the disks to provide support for swapped segments. This module does not modify the process multiplexor, clock manager, or other I/O devices in any way.

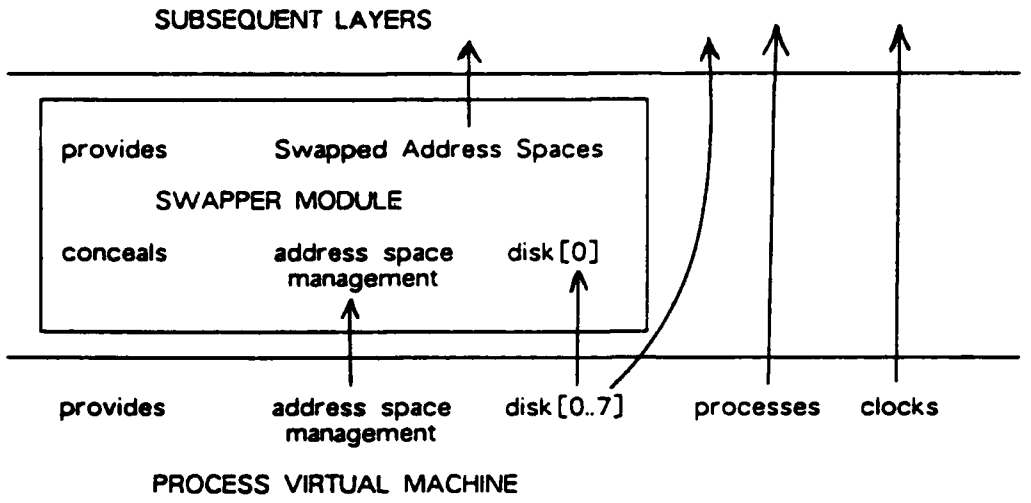


Figure 3-2: Incremental virtual machines in FAMOS

A module that builds upon certain features of the underlying machine level, must conceal those features from subsequent layers to protect its own integrity. For example, a process scheduler can only guarantee fair scheduling if it has *exclusive* access to the CPU's "process state register" (e.g. VAX's Process Control Block Base register). If some other system module can switch process contexts without the scheduler's consent, the scheduler can fail to provide the service it promises.

A strongly typed, modular program defines access to a program component by specifying the *scope* of the component's name. Only those program modules in which the name is *visible* can invoke the name (and thus use the feature. However, existing and proposed scope control mechanisms are inadequate for describing *exclusive access* to some *portion* of a module's features, because

- One module cannot enclose a *portion* of another module.
- An identifier's scope specification is *distributed* over most of the modules that comprise the scope.
- Exclusive access cannot in general be verified by the compiler.

The **acquires** clause I shall propose is a *localized* specification of exclusive access, that allows a module to *conceal* an identifier within its boundaries *without* enclosing the module that provides the feature.

We shall begin the discussion by defining a simple notation for modules, that captures the common properties of existing scope control mechanisms, and by describing an access scenario to be specified by scope control. Then, we shall examine how exclusive access can be specified with existing notations, by considering two cases:

- The module needing exclusive access encloses the module that defines

the feature needed. This strategy contorts and obscures the program structure by cluttering the enclosing module with the names of *all* of the enclosed module's features, most of which it does not need.

- The module needing exclusive access imports (requires, uses) the needed feature. This strategy allows a change to a distant module (requiring the name when it previously did not) to destroy the exclusive access property without being detected by the translator.

Next we shall define the *acquires* clause, specifying its affect on the scopes of the identifiers it names, and illustrating its meaning with several examples. Finally, we shall program the access scenario using the *acquires* clause to specify exclusive access, and see that it satisfies the goals of the problem.

3.2.1. A Conventional Scope Control Mechanism

To simplify the following discussion let us assume a data abstraction language with a **module** construct for dividing programs into statically nested, closed scopes, and two clauses, **provides** and **requires**, for allowing names to be visible in more that *one scope. That is,*

- A name is visible in the scope where it is declared.
- A name visible within a module is visible outside that module *if and only if* that module **provides** that name.
- A *name visible immediately* outside a module is visible within that module *if and only if* that module **requires** that name.

This very simple language for modularization lacks many of the elegant properties of existing and proposed module constructs, but it has two properties that they all share: closed scopes are strictly nested, and visibility of names is regulated at the boundary of each closed scope.

To determine the scope of an identifier one must examine the defining module and its neighbors, to determine what modules are *reachable* from the defining module via *provides* and *requires* clauses that name the identifier. For a particular module to have the use of an identifier, there must be a path from that module to the defining module. More formally, we define the following relations:

Di(M)	Identifier i is declared in module M.
Ri(M,N)	Module M immediately encloses module N, and N **requires** i
Pi(M,N)	Module N immediately encloses module M, and M **provides** i
ADJi(M,N)	Pi ∪ Ri (adjacency)

A program **provides** and **requires** correctly *if and only if*, for each i,

- there exists a unique M such that Di(M), and
- the graph defined by ADJi forms a tree with root M.

This formulation implies that an identifier may not be both required and provided by the same module, and that all the modules in the base set of the relation are reachable from the declaring module. The scope of i is the basis set of ADJi, that is,

The scope of i, where i is declared in module M, is { N | $ADJi^*(M,N)$}

3.2.2. An Access Control Problem

In incremental machine design, one often finds that a given module needs exclusive access to only a subset of the facilities provided by the underlying virtual machine. This most often happens when a set of facilities that appear to be independent of one another, according to their specifications, turn out to be coupled in their implementation. For example, two different I/O devices might have no *logical* relationship to one another, but might be coupled through shared use of interrupt hardware and software. Therefore, a single module that encapsulates the design of the machine's I/O subsystem, would *provide* the names of all the I/O devices present on the machine.

As a basis for analyzing scope control mechanisms, we shall consider a system consisting of

- A hardware module VM0, that provides the hardware devices "terminal", "disk", and "clock";
- A Clock Manager module that uses the clock to provide a time-stamping procedure.
- A File Manager module that uses the disk to implement files.
- A Graphics module that uses terminals to display graphical images.
- A system of modules that use clocks, files and graphics.

The hardware devices must all be declared within VM0 for some unspecified but unavoidable reason. Each intermediate module needs exclusive access to the device it uses, to maintain its integrity. Each intermediate module is programmed by a different person, who is only dimly aware of the specifications of any modules his program doesn't use. There are no hidden dependencies among the three intermediate modules. Informally, the virtual machine layers of this system would look like

Subsequent Layers
use
Time,Files,Graphics

Clock Manager	**File Manager**	**Graphics Manager**
provides	*provides*	*provides*
Time	Files	Graphics
conceals	conceals	conceals
Clock	Disk	Terminals

VM0
provides
Clock,Disk,Terminals

An adequate scope specification for these modules would have the following properties:

- A module exporting an identifier should NOT control access to that identifier in the surrounding text. (A module should work properly regardless of how the facilities it provides are used.)
- A module needing exclusive use of an identifier should be able to declare that need in its specification, and have that declaration enforced by the language. (The declaration should be *localized* and attached to the program unit most affected.)
- A module should control the scope of only those identifiers that are relevant to it purpose. (It should only know what it needs to know.)
- The need for exclusive access should not contort the program structure unduly.

3.2.3. Concealment By Containment

One way a module can be assured of exclusive access to a feature is by *containing* the module that provides the feature. Abstractly, if module A is a virtual machine providing features F, G, and H, and module B uses feature F to implement improved feature I, then the modules would be composed as in figure 3-3.

```
Module B
   provides G, H, I
        Module A
           provides F, G, H
           <<implementation of F, G, and H>>
        end Module A
   procedure I = . . . << invocation of F >> . . .
   << other implementation details of B >>
end Module B
```

Figure 3-3: Concealment by Containment

Unfortunately, this design does not adequately document the fact that module B has not invoked resources G and H at all. Not only did B have to specifically export two items (G and H) that it didn't use, but presumably the specifications of G and H had to be transmitted as well, showing that in fact B did not modify them.

Although the clumsiness of this method may not seem burdensome in any particular case, it has serious implications for the overall structure of a system. Essentially, it calls for a new virtual machine level for each management module. This in turn imposes a total ordering on system features, some of which have no intrinsic relation to one another. The access control problem defined earlier would have to be programmed something like figure 3-4.

```
module VM3
   provides graphics, files, time
        module VM2
           provides terminal, files, time
                module VM1
                   provides terminal, disk, time
                        module VM0
                           provides terminal, disk, clock

                           << defines device hardware >>
                        end module VM0
                   proc time = . . . << invokes clock >> . . .
                end module VM1
           type files =
                < type definition using disk >
        end module VM 2

   <<implementation of graphics facilities using terminal >>
end module VM3
```

Figure 3-4: Multi-level Concealment by Containment

The modules VM1, VM2, and VM3 have been totally ordered by containment, in

order to achieve concealment. This ordering is clumsy, and unnecessary, since "graphics", "files", and "time" have nothing to do with one another. One could imagine with even a moderately complex operating system, that the provides clauses could be clogged with unrelated names. Furthermore, whenever an inner module was modified to add a new feature, each subsequent module would have to add the name of that feature to its **provides** clause.

This difficulty comes about when two or more virtual machine features at a given level appear to be independent, in their specifications, but in fact are related in their implementations. They must be declared in a single module, because they are related, but can be used by different modules, because they are made to behave independently. However, a module that needs only one such feature cannot control its scope without controlling the scopes of all the objects exported from the same module.

3.2.4. Concealment without Containment

An alternate design for modules A and B of the previous section would place module A and module B side by side, and have module B import only the resources it uses, as in figure 3-5. This design allows module B to leave undisturbed those features of module A that it doesn't need; however, module B has not concealed resource F, either! Indeed, it may not be possible to verify the correctness of module B without external proof that resource F is not used elsewhere.

```
Module A
   provides F, G, H

   << implementation of F, G, and H >>
end module A
```

```
Module B
   requires F
   provides I

   procedure I = . . . << invocation of F >> . . .

   << other implementation details of B >>
end module B
```

Figure 3-5: Importing as needed

To specify exclusive access, one must arrange the *provides* and *requires* clauses to give the access needed, and *refrain from* providing or requiring the name anywhere else. The access problem we've chosen would be specified as in figure 3-6.

```
Module VM1
   provides graphics, files, time

        module VM0
           provides terminal, disk, clock

           << definitions of device hardware >>
        end module VM0

        module Clock Manager
           provides time
           requires clock

           proc time = . . . << invocation of clock >> . . .
           end module Clock Manager

        module File Manager
           provides files
           requires disk

           type files =
                < type definition using disk >
        end module File manager

        module Graphics
           requires terminal
           provides facilities for graphics displays

           <<implementation of graphics using terminal >>
        end module graphics
end module VM1
```

Figure 3-6: Distributed Specification of Exclusive Access

This program specifies that, for example, module *Graphics* has exclusive access to the identifier *terminal*, by

- *requiring* the identifier into the module *Graphics*,
- *not* requiring it into *Clock Manager* or *File Manager*, and
- *not* providing it out of the module *VM1*.

Such a specification is inadequate because it imposes an unstated constraint on the clock manager and file manager, namely that they *not* require "terminal". Furthermore, if the programmer of either module overlooks or ignores the constraint, and violates it by *requiring* the identifier, the violation will not be detected. This exclusive access specification is therefore unsatisfactory because it is *distributed* over several modules that it should not involve, and *unenforced* by the language system.

3.2.5. The *acquires* Clause

To resolve these difficulties, I propose a simple addition to the set of ways one can transport a name across a module boundary: the **acquires** clause. Informally, a module that **acquires** an identifier "requires exclusive access" to it. For example, we would rewrite figure 3-3 as in figure 3-7.

```
Module A
   provides F, G, H

   << implementation of F, G, and H >>
end module A
```

```
Module B
   acquires F     << note change >>
   provides I

   procedure I = . . . << invocation of F >> . . .

   << other implementation details of B >>
end module B
```

Figure 3-7: Acquiring as needed

Here Module A provides unrelated resources F, G, and H. Resource F is "parcelled out" to module B for management. The acquires clause specifies that no other module (outside A) may access resource F. (Remember that the reason F is declared in A and not B is that the *representation* of F is coupled to other resources in A.)

To make the acquires clause testable and enforceable, we must specify precisely the conditions under which it may be used, and its meaning under those conditions:

> An identifier that would otherwise be visible in a scope may be acquired by at most one module in that scope. Furthermore, it may only be acquired when exclusive access can be guaranteed.

Exclusive access can only be guaranteed if the following conditions are met:

- The module enclosing the scope neither **requires** nor **provides** the identifier. This would risk use of the identifier outside the enclosing module.
- No other module in the scope **requires** or **acquires** the identifier.
- No other program unit (such as a procedure or declaration) in the scope uses the identifier.

More formally, we redefine the scope rules for the language with the following relations:

Di(M)	Identifier i is declared in module M.
Ri(M,N)	Module M immediately encloses module N, and N **requires** i
Pi(M,N)	Module N immediately encloses module M, and M **provides** i

Ai(M,N) Module M immediately encloses module N, and N acquires i

ADJi(M,N) Pi ∪ Ri ∪ Ai (adjacency)

A program provides, requires, and acquires correctly *if and only if*, for each i,

- there exists a unique M such that Di(M),
- the graph defined by ADJi forms a tree with root M, and
- $\forall X,Y,Z\ Ai(X,Y) \wedge ADJi(X,Z) \Rightarrow Y=Z$

That is, if Y acquires i from X, then Y is the only module that obtains access to i from module X. The scope of i is the basis set of ADJi, excluding those modules from which i has been acquired. That is, the scope of i, where i is declared in module M, is

$$\{ N \mid ADJi^{*}(M,N) \wedge \forall P \sim Ai(N,P) \}$$

3.2.6. Examples

Let us consider some examples of how the **acquires** clause might be used:

Suppose an identifier declared in a scope is **acquired** by a module declared in the same scope. The visibility of that identifier is limited to the inner module.

```
Module main
   var C
      module B
         acquires C
         << The scope of C is { B } >>
         end module B
<< C is not visible here >>
end module main
```

Suppose an identifier is provided to a module by one of the modules declared within it, and acquired by another. The visibility of the identifier is limited to the two inner modules.

```
module main
        module C
           provides D

           var D

           << D may be used within this module >>
        end module C

        module B
           acquires D

           << The scope of D is { B, C } >>
        end module B
<< D may not be used here >>
end module main
```

One might argue that using an acquired identifier *inside* the providing module violates the exclusive access granted to the acquirer. However, if such use affects the external behavior of the object, it will be documented in the providing module's specification. The acquiring module only "knows about" the outermost specification for the identifier, and only requires exclusive access relative to that specification.

Suppose an identifier is **provided** by several modules enclosing its declaration, then **acquired**. The identifier may be used within any of the providing modules, and within the innermost acquiring module.

```
module Main
        Module Prov1
           provides X
                Module Prov2
                   provides X

                   var X
                   << X may be used here >>
                end module Prov2

           << X may be used here >>
        end module Prov1

        module Acq1
           acquires X
                module Acq2
                   acquires X

                   << The scope of X is
                              { Prov1, Prov2, Acq2 } >>
                end module Acq2
           << X may not be used here >>
        end module Acq1
end module Main
```

To summarize, the essence of the **acquires** mechanism is that the identifier is *allocated* from the outermost **providing** module to the innermost **acquiring** module. That innermost module is guaranteed exclusive use of the identifier. The allocation, furthermore, is checkable at compile time.

3.2.7. The Access Problem, Revisited

Returning to the example of the three device managers, we may compose them using **acquires** as in figure 3-8.

```
Module VM1
   provides graphics, files, time

      module VM0
         provides terminal, disk, clock

         << definitions of device hardware >>
      end module VM0

      module Clock Manager
         provides time
         acquires clock

         proc time = . . . << invocation of clock >> . . .
         end module Clock Manager

      module File Manager
         provides files
         acquires disk

         type files =
              <  type definition using disk >
      end module File manager

      module Graphics
         requires terminal

         <<implementation of graphics using terminal >>
      end module graphics
end module VM1
```

Figure 3-8: Multiple Acquiring Modules

Note that each module needing exclusive access has a localized specification of the need, enforced by the language. For example, if the File Manager's specification were changed to **require** "terminal", the translator would detect the conflict with Graphics, which **acquires** "terminal". Furthermore, the nesting structure of the program generally reflects the natural compositional structure of the system. Virtual Machine 1 is composed of an underlying virtual machine (VM0) and three modules, each of which conceals the features it uses.

An aside about aliases: Note that each of the modules **provides** a facility built upon the **acquired** resource. If the resource were only *required*, this would create an alias for the resource. Languages that attempt to prevent aliases, such as Euclid, would forbid the latter construction. However, by acquiring an identifier instead of requiring it, a module actually removes that name from the surrounding environment, thereby removing the alias problem.

3.3. Environments as Modules

In section 3.1 we developed the notion of an environment by looking at operating system protection environments and language run time systems. In this section we shall clarify the roles of both language systems and environments in operating system descriptions.

One purpose of a systems implementation language is to provide, for each environment, a programming notation suited to the facilities available in that environment. Many of these facilities correspond to features of traditional high-level languages. These high-level language constructs pose the following dilemma:

- A language construct *can* provide a valuable abstract notation for using an operating system facility.
- Incorporating the construct in a language, before the system is designed, prematurely commits the system to provide the facility.
- The price of not providing the language construct is a significant loss of abstraction in the system description, and lost ability to verify or enforce system properties based on syntactic structure.

To resolve this dilemma, I propose to design operating systems and their implementation languages simultaneously. Then, in each environment, the operating system shall provide components of the implementation language support system.

An environment specification in such a system defines the interface between the language system and the operating system. We shall study an example (from the language Euclid) where the language allows a program to provide part of its own run time support (dynamic storage management) by giving specifications for a support module (a zone module) to be provided by the program and used by the language system.

The specification method used for Euclid zones may be applied to any ordinary run-time facility, such as synchronization or exception handling. However, most environment features require interaction between the compiler or linker and the operating system, such as when the compiler places a variable in a section, linked into a segment, and generates instructions that use virtual addresses to access the variable. To accommodate these features, we shall explore the view that a compiler is a type manager for the abstract type "program", which it implements using the types *section* and *segment* and the primitive types of the execution environment. This view will allow us to represent the compiler, linker, and environments as ordinary modules in system descriptions.

3.3.1. Language Systems in Operating Systems

A programming language is both a design notation and an implementation tool. As a design notation, it must *fit* the problem domain. It must give the system designer the ability to describe the system in a way that emphasizes the important attributes of the system. As an implementation tool, it must be precise and complete. That is, it must describe everything about a system, exactly. A good language, both for design and implementation, will allow the programmer to juxtapose pieces of information that are interrelated, so that the reader of the design can study that interaction without having to sift through lots of unrelated material. In order to present the important material, the language must *conceal* irrelevant detail. This emphasis and concealment is often called *abstraction*.

Languages support abstraction in two ways: by providing built-in abstractions, (e.g. infix arithmetic operators, semaphores), and by providing tools for constructing new abstractions (e.g. type definitions, macros). With an appropriate set of primitives and good construction tools, one can construct a set of abstractions that fit the problem domain.

However, sometimes the price of abstraction is loss of transparency or efficiency. Certain programming techniques, such as performing arithmetic on pointers, are impossible to achieve in most strongly typed languages. Languages that replace the goto statement with rich vocabularies of control constructs, such as Bliss-11, are usually inadequate for expressing certain exception-handling techniques. Retaining the goto statement often results in distributed overheads, either by preventing certain optimizations or by making the cost of discovering them prohibitive.

A language with good *construction* tools might still fail to support certain abstractions because of inadequate *primitives*. For example, a language without synchronization primitives might not be able to support good synchronization abstractions. Even if the proper code sequences can be generated to implement, say, monitors, the language cannot provide the static checking needed to make sure that monitor entry and exit are properly nested.

Recent systems implementation languages can be classified in two groups:

- Transparent languages, such as Bliss and C, which emphasize access to the machine and very good code generation, but do not enforce data types
- Strongly typed languages, such as Euclid, Modula, and Gypsy, which attempt to provide methodologically sound, type-safe abstractions corresponding to the features of the underlying machine. Typical abstractions provide multi-tasking, synchronization, and device communication.

Although transparency is often a desirable, even necessary language property, I reject existing transparent languages for large system construction for their lack of strong type facilities.

The advantages of the strongly typed approach are three: the compiler can use special knowledge about the facility to generate better code, it can automatically generate a management data structure customized to the program being compiled, and it can perform static checks on the proper use of the facility. Modula, for instance, by providing a particular form of *monitors*, was able to ensure mutual exclusion without any run time overhead at monitor entry or exit. Using a transparent language to construct monitors requires the programmer to check for himself that the entry and exit procedures are invoked at the proper times.

The disadvantages of built-in facilities are:

- Hidden structure: such facilities usually require non-trivial run-time support, which is normally excluded from system descriptions as a "mere implementation detail".
- Lost transparency: to protect the integrity of the built-in facility, the language must prohibit any other access to the machine features upon which it is built.
- Imposed design constraints: as mentioned in section 2.2.1, providing a particular synchronization construct in a system implementation language constrains the class of systems that can be built in that language.

Nevertheless, an operating system component *should* be written in a notation that expresses the abstract properties of the facilities it uses. For instance, if a message system is implemented as a set of cooperating processes, it should be written in a language with facilities for multiprogramming and synchronization.

These built-in language facilities embody operating system-like functions. A language that supports dynamic object creation needs a storage manager. A language that supports multiprogramming must have a process scheduler. A language that supports exception handling must have a software error reporting mechanism. All of these are common operating system facilities. Because of the similarity of purpose, we need a way to integrate language support software with operating system facilities, such that there will not be duplication, excessive overhead, or conflict.

3.3.2. Incremental Programming System Design

To integrate an operating system and its implementation language, they should be designed together. Each virtual machine level of an incremental operating system design defines one or more execution environments for use by subsequent levels. For each environment, the implementation language should provide constructs that embody the abstract properties and proper use of the available facilities. For example, a programming language construct for exceptional condition handling might be the methodologically "right" way of using software trap vectors.

Starting with a language for sequential programming with abstract data types, the

system designers would add facilities for multiprogramming, exception handling, dynamic objects, etc, as the corresponding facilities were being designed for the operating system. Specifically, each operating system facility would be designed with a particular language construct in mind, so that the virtual machine level which provided the facility could support programs that used the construct. Each execution environment would support a dialect of the implementation language tailored to the facilities of the environment. In this way, each system level could be programmed in a notation fitting both the requirements of the level and the facilities available.

Looking back at FAMOS, it could have been designed as follows:

Level	Facility	Language Feature
VM5	Waiting Lists	Semaphores, Path Expressions
VM4	Process Management	Multitasking
VM3	Address Spaces	Overlaying, "Named Common"
VM2	Space Allocation	Dynamic Objects
VM1	Interrupt Masking	Critical Regions
Hardware	Interrupt Vectors	Exception Handling

Figure 3-9: FAMOS as a Language Support System

Actually implementing a separate language for each system level would be outrageously expensive. Instead, in chapter 7.4 we will discuss tools such as preprocessors, parser generators, linkage editors, and compiler compilers, that make the costs manageable by sharing most of the translation software among levels.

Next we shall consider how to specify the interface between the operating system and the language at each system level.

3.3.3. Specifying the Module Interface

Developing systems implementation languages concurrently with developing the systems they are to implement, will only lead to serious delays in both efforts, unless the interface between the two is carefully specified. The interface must provide the same quality of separation between the compiler and the environment support facilities as between one operating system component and another.

Therefore I propose the following technique:

> Each language dialect shall give a specification for the minimum features it demands of the execution environments it will use. This specification will be in the form of an *environment module*, which shall define the programming interface between the language system and the operating system.

To see what form this specification might take, we shall look at the dynamic storage management facility of the Euclid run time environment, which is specified just this way. Euclid allows programs to manage their own dynamic storage pools, called **storage zones**. A dynamic record type in Euclid is called a **collection**; two

records from different collections have different types, even if the two collections use the same base type to define the structure of individual instances. The declaration of a collection may optionally name a storage zone module. If it does, all members of the collection will be created out of storage allocated from that zone.

Euclid gives a set of abstract and syntactic specifications for a set of standard procedures that every storage zone module must provide. However, these specifications are only a minimal set of requirements. The module providing the storage zone may provide any additional services that are appropriate, such as statistics gathering facilities or "storage low" warnings. The user of a collection invokes language operators **new** and **free** to create and delete objects; those routines (presumably just a few in-line instructions) call the zone manager to obtain and release space, and invoke the initialization and finalization code for the collection's base type.

Whenever an object is created, a type transformation take place. The Euclid language uses the type "allocation unit", implemented by an ordinary user module, to implement the language feature "dynamic record". Another interesting type transformation takes place within the zone manager, between storage units, addresses, and pointers. A Euclid zone manager must somehow declare a variable which occupies the storage it is to manage, then break up that variable into allocation units according to the demands of the zone users. To facilitate this, the language defines a type **storage unit** which has no operations defined on it, and no distinguishable values. However, there is a function which maps an array of storage units into the **address** of the first element. The type **address** is a subrange of the integers, allowing arithmetic on addresses. There is also a guarantee that addresses and pointers have the same standard representation. This facility allows the zone manager to declare a vector of **storage units**, compute the address of any position in the vector, and create a pointer to it.

To summarize, the Euclid storage zone mechanism has the following properties:

- The type conversion between allocation unit and dynamic record is protected by the compiler
- The storage zone manager is written in essentially the same language as the user program, and can be included like any other program component in the overall system description.
- A program may invoke the zone manager directly, for instance to ask how much space is available, or indirectly, via the language features **new** and **free**.
- The language system uses the storage zone *mechanism* to represent variables, without imposing significant constraints on storage allocation *policy*.
- The language provides special-purpose types to aid in the implementation of zones.

- The verification of the programs which *use* the storage zone can be separate from the verification of the zone manager.

The storage zone is the only environment feature that a Euclid program is allowed to supply for itself. However, the same approach ought to be suitable for process management, synchronization, exception handling, address space management, capabilities, and message systems. To generalize from the zone manager, observe that for each language construct requiring environmental support, there are a set of types, variable and procedures that characterize the interaction between the compiler and the environment. Each type is either implemented by the compiler and used by the environment manager (e.g. storage unit for zone manager, state vector for process manager), or implemented by the environment manager and used by the compiler (e.g. allocation unit for zone manager, waiting list for synchronization).

A language specification for an environment facility, therefore, would include:

- The specifications of the types the facility must provide
- The specifications of the operations which must accompany the type
- Those compiler-implemented types which the environment manager will need to construct the types it provides

For zone managers, Euclid specifies the names of the allocation procedures, including the types of their parameters, and provides the types **storage unit**, **address**, and **pointer** for constructing storage managers. A language which supported cooperating processes would specify the routines any process manager must provide, and define the type **state descriptor** which would contain that portion of the execution state of a process for which the compiler was responsible. That type would support operations like "load" and "unload", for installing it as the currently executing program, without concerning the operating system designer with details of which registers to save and restore. The process manager would then construct the type **process descriptor** by combining the type **state descriptor** with whatever other information was appropriate to the operating system design. The operations provided to the language system would then deal in process descriptors and state descriptors.

The types and procedures would not necessarily have to be implemented at the time the compiler was generated; they might simply form a database which the compiler used for code generation. In that way, the development of the compiler and the operating system could proceed independently. These issues will be discussed more in Section 7.4.

By specifying the boundary between the language system and the operating system in terms of types and procedures, we can separate those design decisions having to do with code generation and optimality, from those dealing with environment management and operating system structure. In particular, the translator can provide

a *mechanism* for connecting an object (e.g. process) to its container (e.g. process descriptor) without imposing *policy* about how that container is to be implemented.

3.3.4. Compilers, Sections, Segments, and Systems

So far, we have discussed only specifications for conventional run time facilities. However, most of the features of an environment require interaction between the language system and the environment manager during translation, such as when the compiler places a variable in a segment, then selects appropriate instructions and virtual addresses to implement operations on the variable. Compilers, linkers, and loaders, in present technology, are *processes* which transform a source-language program into a running system. If we view them instead as *type managers* which provide the concrete representation of an abstract program, we can use abstract data types to describe the translation system's role in an operating system. Then we can sketch the specification of each kind of environment feature.

We take as our "atomic unit" of a compiled program, the *section*. A *section* can be an arbitrary sequence of machine language instructions and data, whose value has been determined by a compiler. A section might represent a variable, or a procedure, or several variables and procedures, or whatever the compiler cares to produce. However, it must correspond to a *contiguous* sequence of memory locations, so that the linker can ignore any internal divisions.

Let a compiler is a type manager for the abstract type **program**, where (for the time being) we define **program** as the unit of source code given to a compiler for translation. The concrete representation of a program shall be a group of **sections** containing the code and data, plus whatever information about the program must be recorded in the environment management data structures. The **sections** a compiler creates for a program can be placed in different environments.

Let a linker be a type manager for the types **segment** and **section**. A segment is an administrative unit of the operating system's virtual memory manager. The internal layout of objects in a segment is not known to the memory manager, although the segment might be divided into pages, at run time, for efficient storage administration. The linker concatenates a sequence of sections into a segment, thus fixing the displacements of identifiers within segments. The linker could provide a global symbol table mapping logical names into logical addresses.

An **environment** is a module or type, and is implemented by the operating system. It provides an implementation of each of the conceptual environment components described in section 3.1 including:

- A logical memory, perhaps of type **set of segment**, defining the accessible logical names.
- A program region, composed of one or more segments, into which the compiler may place sections.

- An address mapping facility, giving access to the virtual memory mechanism provided by the operating system.
- A basic hardware instruction set, defining the primitive types of the target machine, the effects of instructions upon them, et cetera. The machine descriptions Cattell used for automatic generation of code generators [Cattell 78] would be appropriate.
- A set of virtual instructions, including operations on special registers, micro-coded procedures, extended opcodes for invoking kernel procedures, and known entry points to other environments.
- An entry point set.
- Runtime support facilities provided by lower system levels.

We shall discuss in section 3.6 how some of these sets could be generated automatically, during translation, rather than having their contents listed statically. Prime candidates for this are logical memory, known entry points, and the entry point set.

The **environment** module relates to the language system in the same way that Euclid's **zone managers** do: the language defines the minimal set of facilities any **environment** must provide, and accepts as an **environment** any module which satisfies the requirements. The module may provide other facilities as well; also, there can be many different **environment** modules, with different implementations.

We are accustomed to seeing compilers and linkers as "host machine" software. *Environments are, too!* An environment must accumulate segments, virtual instructions, entrypoints, and other program representation information, and disseminate the information to the compiler. We will discuss in section 3.6 how this can be done.

The translation support modules could be specified something like as shown in figure 3-10. The linker and compiler modules can be thought of as a "standard prelude" to a system description. The type **section** is managed by a particular instance of the type **segment**. The types **procedure, variable, and task**, each take as a generic parameter the environment in which the particular instance will be located. The system description itself begins with a module which satisfies the syntactic specifications for an environment.

```
module linker
     provides type segment
              type section
              type logical name
              type logical address
              module symbol table

module compiler
     requires linker
     provides type procedure<E:environment>
              type variable<E:environment>
              type task<E:environment>
              et cetera

module BaseMachine       -- an environment
     requires segment, et cetera
     provides var InstructionSet
              var LogicalSegmentSet
              var ProgramRegion
              module Address Map
              module Capabilities
              module Entrypoints
              module Synchronization
```

Figure 3-10: Specifications for Translators

In this "modular decomposition" of the language support task, we have not said very much about how control flow will pass through such a type description to actually create a running system. We will discuss that question at length in Section 7.4.

3.4. Type Management in Operating Systems

An operating system is responsible for decomposing a single object, *The Machine*, into useful components which behave as though they were independent of one another (most of the time). Consequently, many of the type managers that appear in operating systems use shared variables, descriptor pools, and management sets to implement types whose instances appear, from the viewpoint of their users, to be independent entities. Those users are often located in separate environments. Now that we can describe environments, we can talk about programs whose components reside in different environments. We approach the subject by identifying four classes of data type management commonly occurring in operating systems. For each class we examine how *coupling* between type instances, via shared data, affects the ways in which instances of such types might be scattered across several environments. To illustrate the difficulties involved, we review the design of the virtual clock manager in FAMOS.

3.4.1. Kinds of Type Managers

A type manager is the source-language program unit that implements a type. One dimension along which styles of type management vary is the degree to which the type manager controls the instances of the type. I see four major bands in the spectrum of possibilities:

1. Value types: The essential aspect of the type is its abstract value, and the set of abstract functions available involving values of the type. The primitive types of conventional programming languages are representative; the type manager for each of them merely supplies functions over values of the type.

2. Object types: The essence of the type is not only its abstract value, but also some sense of its state. For example, a stack is an object type. The primitive operations defined on a stack all take into account its previous value, i.e. its state. The notion of copying a stack variable is not ordinarily meaningful. When a stack variable is first created, it is in a distinguished state, *uninitialized*. None of the stack operations may be applied to it until the "initialize" operation sets the stack's state to *empty*.

3. Monitored types: The type manager has continuous access to all instances of the type. It may operate on instances of a type asynchronously with respect to operations invoked by owners. In contrast, a type manager for a value type or an object type has no need to keep track of the set of existing variables of that type. For value types, the type manager never has to access the type-user's objects at all. (Ideally, the language system provides assignment and equality operators automatically.) For object types, the type manager must examine the object in order to correctly compute its next value, but once an operation is complete the type manager relinquishes access to the object of the operation. Garbage-collecting storage managers implement the type pointer as a monitored type. The storage manager keeps a reference to every pointer variable of the user's program, so that the garbage collector can determine which storage units are in use and which can be reclaimed.

4. Allocated types: An allocated type has a limited number of instances, usually because its instances *are* resources, or *use* resources. The type manager declares how many instances of the type there will be, and provides operations to allocate instances to users on request. Obvious examples are hardware devices such as magnetic tape drives and Arpanet connections; less obvious ones are user-level "jobs" in a timesharing system, and operating system table entries. An allocated type is different from a monitored type in that the manager of a monitored type does not *control* the number of instances of the type, although it must have access to all of them; an allocated type manager actually *determines* the number of type instances.

Some of the differences between these classes of types can be seen in the parameter mechanisms that are applicable to them. Value types may be passed by any mechanism available: value, reference, name, copy, result, deferred evaluation, etc. Object types may be passed by name, by reference or by read-only; copying of the object would be disallowed. Monitored types could be passed by reference, but only if the type manager avoided (or documented) asynchronous changes to the *abstract* state of its instances. Allocated types cannot be passed by reference if

the resource can be deallocated from the user, because it must be possible to delete all outstanding references to the resource. Instead of passing the resource object itself, the right to access an allocated object must be recorded in an instance of an *object* or *monitored* type, analogous to a capability. Then, that capability could be passed as a reference parameter.

3.4.2. Coupling

Two variables are *coupled* if an operation on one of them can change the value of the other [Parnas 78a]. The extent to which type instances are coupled limits the ways in which they can be distributed among different environments. Coupling among instances of a type can occur at at least three levels:

- The abstract type specification may refer to an abstract object that is shared among the type instances, e.g. a tree that is shared by all of its nodes.
- The *representation* of instances may be coupled, as when some property of an object is represented by its presence in a linked list of objects having that property.
- The instances may be implemented using a shared resource, whether allocated (e.g. memory) or multiplexed (e.g. CPU)

The last two forms of coupling do not couple the *abstract* objects, but only their *representations*. Furthermore, the last form would not even show the coupling in the source-language representation of the program. Two variables created from the same storage pool have coupled representations, in the sense that they are both components of the shared variable "vector of storage"; however, the storage manager makes sure that during the lifetime of a variable, changes to the storage vector do not cause changes in the value of the particular variable.

For each kind of type management we can examine how coupling might be manifested:

- *Value* type instances are never coupled, since replacing one value with another in one variable does not disturb any other variable.
- *Object* type instances can be coupled either by representation, or by resources, but not abstractly. The abstract state of an object may not be changed except by an explicit operation on that instance.
- *Monitored* type instances are coupled at some level. For example, a wakeup operation on a waiting set changes the abstract state of some process from "blocked" to "ready". A process scheduler may reallocate a processor from one process to another, changing their concrete states from "running" to "not running" and vice versa, without affecting the abstract states. A page of memory forced out of primary memory to make room for another, is coupled to its preemptor by resource usage.
- *Allocated* type instances are generally coupled to the allocation data structure, and often through resources, but may or may not have

coupled representations or abstractions. Two magnetic tape drives may be administered by a central allocator, but are otherwise independent (unless they share a controller or trap vector).

3.4.3. Type Managers in Multi-environment Programs

The extent to which instances of a type are coupled determines the options available for distributing them among multiple environments. The environments in which they occur must be be sufficiently connected to support the coupling. For example,

- A value type is never coupled, so its instances can be freely scattered. Each invocation of an operation may be an inline expansion of the operation text. If many invocations of an operation appear in the same environment, a copy of the operation may be made into an executable procedure and placed in the environment.

- An uncoupled object type may be treated much like a value type, insofar as simple operations are concerned. However, since object types can be passed to procedures only "by reference", an object type instance can only be passed between environments if the protected procedure mechanism supports by-reference parameters.

- A coupled object type instance, and the objects with which it is coupled, must all be addressable simultaneously, but only for the duration of each operation on the instance. This implies that such an operation can be executed in any environment that can access the particular objects that are coupled.

- All instances of a monitored type must be located where they are permanently accessible to the operations of the type manager. Generally, this implies that either such objects are all located in a single program region, with references to them handed out to their owners, or that all objects are located in a known set of segments, which are made accessible to whichever environments contain instructions that must address them.

- An allocated type is constrained by the origin of the resources it represents. Whatever environment has access to the basic resources must also contain the declarations of the allocated type objects. For example, device register variables on a PDP-11 must be declared in an environment that can address the physical memory segment containing those device registers.

Choices among the options above cannot be automated with present technology, nor should they be. The choices can have significant impact on performance and on system structure, so we seek instead the tools to let the programmer describe the bindings he wants. The constraints above will then allow automatic validation for the proposed bindings.

We next examine in detail how monitored types were used in FAMOS, to gain a deeper understanding of the problems they pose for multi-environment systems.

3.4.4. Monitored Types in FAMOS

Most of the facility managers in FAMOS could be characterized as managers of monitored types. In this section we begin by describing the clock manager in that fashion, and abstracting the general properties of the FAMOS facility managers that make implementation using monitored types appropriate. However, the clock manager was actually implemented as an allocated type manager, as were most of the managers of permanent objects. We shall discuss the actual implementation of the clock manager, concluding that the discrepancy is due primarily to inadequacies in the program development facility. Since FAMOS was implemented in an untyped language, we first try to remedy the inadequacies by programming the clock manager in a strongly typed language, Euclid. We see that monitored types can be programmed easily in that language, provided that the type instances are to be created and deleted dynamically, and management sets are implemented as linked lists. However, many of the monitored types in FAMOS have only permanent instances, and are implemented as static vectors. We attempt to program FAMOS clocks this way, borrowing notations from several languages. We discover that such a module places substantial demands on the program translation facility. In particular, it must be able to elaborate arbitrarily complex type initialization procedures during translation, and must allow those procedures to invoke the static storage allocator of the translator itself.

The FAMOS clock manager implements an abstract type **virtual clock** which embodies the timing facilities provided by the clock module. A virtual clock is specified abstractly as an interval timer, with operations *start*, *stop*, and *settime*. The clock manager uses a single hardware interval timer to implement all the virtual clocks. The hardware timer is set to interrupt when the first of the running clocks is due to expire. When the interrupt occurs, the clock manager notifies the user of the clock, and resets the timer for the next most imminent alarm. The clock manager must be able to access any instance at any time in order to determine when to send a wakeup signal to the owner of the clock, but the correctness of the implementation in no way depends on how many virtual clocks are to be managed. Therefore the type **virtual clock** can be considered a monitored type. To implement continuous access, the clock descriptors are all located in the clock manager's addressing environment. Conceptually, each module that declares a virtual clock variable refers to it by a pointer into the clock manager's address space. Because the virtual clocks are all static objects, the pointers are load-time constants. The clocks occupy consecutive storage locations in the clock manager's address space. In one implementation they are iterated over by index. In the other, they are linked into a list in order of least remaining time.

From the clock manager we can generalize to a class of type managers for operating system resources, having the following characteristics:

- Each facility is represented to the user as a virtual resource, that

behaves like an idealized version of the actual resource. Examples are clocks, processes, software trap vectors, disks. There can be any number of instances of the virtual resource; the manager conceals how many actual resources there are. The virtual resource can be represented as a monitored type.

- The type used to bind a user to a virtual resource can be any of several kinds. If the binding is permanent, it can be a value type, such as a pointer. The user may copy that pointer as he pleases. If the binding is temporary, but not pre-emptible, the binding type can be an object type, such as a variable containing a pointer. The owner may control access to that variable as he chooses, but may not copy it. That way, the owner may acquire and give up a virtual resource, but that resource can only be taken away as a consequence of an explicit operation invocation. If the binding must be pre-emptible, then the binding type must be a monitored type, so that the type manager can revoke access asynchronously.

- The actual resources being managed are characterized by an allocated type (e.g. processor, storage unit, disk block). The actual resources might be *shared*, *allocated*, or *multiplexed* among the the virtual resources (e.g. clocks, memory, processor respectively). The facility manager would use the actual resources to implement the virtual resources. Since the virtual resources appear as monitored types, pre-emptible bindings are easily implemented. Since the virtual resources are usually monitored rather than allocated, the type binding the user to the virtual resource would not have to be pre-emptible, and could thus be an object or a value type.

Although the clock manager could have been considered a monitored type, it was actually programmed as an allocated type. The clock manager declared a fixed-size vector of virtual clocks, where the size was a compile-time constant. Some of the clocks were statically allocated to certain higher-level modules. The allocation was documented in a public file giving global names to certain clock table indices. Higher level modules referred to clocks only by name, never by index. The types **static address space**, **software interrupt vector**, **protected procedure call stack**, and **process**, were likewise implemented as allocated types.

The allocated type implementation was used in FAMOS to prevent deadlock among kernel services, to protect descriptors from addressing errors in the owner modules, and to accommodate the program development facility in certain minor ways. After discounting problems that are plainly due to that facility, there remain two fundamental flaws in the approach:

- The allocation file must be constructed by hand, and reconstructed for every configuration of FAMOS. Such bookkeeping tasks are prone to trivial but costly errors.

- The allocation file violates Parnas's information hiding principle [Parnas 72b]. Every module that declares a process, for example, must write into its external specification exactly how many it needs, including their names. This sort of design decision is likely to change, leading to changed specifications, and forcing changes to the allocation file.

A better design would be one in which a module using a monitored type could

simply declare a private instance of that type, and let the instantiation code configure the type manager to accommodate the new instance. The module specification for the owner of the instance would state only that it *used* the type, and not the number of instances (if any) it created.

Since FAMOS was programmed in an untyped language, we must ask whether simply using a modern strongly typed language would allow one to program a monitored type more satisfactorily. Figure 3-11 gives a Euclid program fragment defining a type virtual clock. A virtual clock is itself an object type, implemented as a pointer to a clock descriptor. The clock descriptor is a monitored type; all instances are linked into a master list headed by the variable *FirstClock*, so that the clock manager can access any descriptor at any time. The clock manager also specifies the storage in which the clocks will be located, by assigning the collection of clock descriptors to the zone *ClockStorage*. Each time an owner creates a virtual clock, the initialization code puts the new clock descriptor into the management set.

```
var ClockManager: module
    exports VirtualClock
    imports ClockStorage

    type ClockDescriptor = forward

    var ClockCollection: collection of ClockDescriptor
            in ClockStorage

    type ClockDescriptor = record
        var NextClock: ^ ClockCollection
        var ClockData: . . .
        end

    var FirstClock: ^ ClockCollection

    type VirtualClock = module
        imports ClockCollection, FirstClock, ClockDescriptor

        var cd: ^ ClockCollection

        initially begin
            ClockCollection.New( cd )
            cd^.NextClock := FirstClock
            FirstClock := cd
            . . .
            end

        end module VirtualClock

    end module ClockManager
```

Figure 3-11: Clock manager, dynamically in Euclid

The key points that make this design work are that the type manager is notified whenever an instance is created, and can carry out whatever bookkeeping may be required, that the type manager is not responsible for allocating type instances, and that the owner of an instance obtains a private, unforgeable name for his clock,

without obtaining access to the whole set of clocks. These features all derive from the Euclid dynamic record facility.

To obtain analogous benefits for the static vector representations used in FAMOS, let us borrow language features from several sources. Alphard supports its full procedure syntax in initialization clauses for permanent objects, and allows constant fields of a record to be computed during its initialization. Algol 68 supports flexible vectors and pointers into vectors. Dijkstra gives a suitable definition of the *HighExtend* operator on flexible vectors [Dijkstra 76]. Figure 3-12 shows a vector implementation of monitored types. Instead of using the NEW function to create a new clock descriptor, it extends the vector of descriptors and binds *CD* (now a constant) to refer to the new one.

```
var clockmanager: module
    exports Virtual Clock
    type Clock Data = . . .
    var Count := 0
    var ClockTable: flex vector [1 .. Count] of Clock Data

    type Virtual Clock (StartTime) = module
        imports Count, ClockTable
        const cd: ref clocktable[]
        initially begin
            HighExtend ( ClockTable )          -- increases length by 1
            Count := Count + 1
            cd := ref ClockTable [ Count ]     -- cd constant henceforth
            SetTime ( C, StartTime )
            end
        end Virtual Clock
    end Clock Manager
```

Figure 3-12: Clock Manager, using Flexible Vectors

Such a structure allows the compiler to know that "cd" is a constant, but still leaves the clock table as a dynamic object. It also describes the initialization of permanent clocks as part of creation, instead of needing a separate call in the startup code. In a system where all virtual clocks were permanent variables, the variable Count would be a constant after system integration, as would be the size of the flexible vector ClockTable. Suppose that a translation system were smart enough to detect that "Count" was going to be a load time constant. It could provide this information to the optimizing phase of the compiler, allowing considerable constant folding. Also, the starting time for each clock could be filled in by the compiler, reducing the amount of initialization code needed at starting time.

However, there are substantial roadblocks to making the above supposition a reality:

- The initialization code for the type Virtual Clock uses procedure calls, flexes vectors, and increments variables. To initialize a permanent clock would require a host machine implementation of the full source language.
- The translator would have to compile the clock manager before compiling owners of clocks, stretch the vector each time it

encountered a clock declaration, then freeze the vector size in time for linking.

- If the vector's elements were a program representation type, such as a process descriptor, the compiler and operating system would have to agree upon the specification of the type, so that the compiler itself could invoke the type definition each time it compiled a process declaration.

A flexible vector implementation of FAMOS segment tables would require even more interaction between initialization code and the translation facility. The segment manager must know the size of each segment in order to allocate enough storage, and must have access to all segment descriptors at all times, but does not care how many there are. Many environments in FAMOS have static logical memories, allowing individual segments to have local names that are compile-time constants. However, the number and size of segments cannot be determined until after the code and data residing in the environment has all been compiled and linked. A dynamic implementation as in the Euclid clock manager, above, would introduce unnecessary overheads due to pointers. A flexible vector implementation would require that the compiler and linker notify the segment manager each time a segment declaration was translated, and also fill in the segment descriptors with the actual segment sizes after they are determined. Furthermore, since the segment manager in FAMOS actually resides in one of the addressing environments it manages, the translator design would have to be very circumspect to avoid infinite recursion.

These difficulties are surmounted in conventional systems by *ad hoc* system generation programs, and ill-structured interfaces between compilers, linkers, and loaders. For example, the Modula process management facility is designed somewhat like the Euclid clock manager in figure 3-11 except that the process management set is not linked together until system startup. The compiler creates a process descriptor each time it compiles a process declaration, and initializes it with most of the initial state description for the process. However, the "main program" must explicitly call an initialization routine for each process, during startup, at which time the process is linked into the management set. This design violates the information hiding principle just as the FAMOS allocation files do, by requiring the main program to know the name of every process.

In section 3.3.4 we developed a type model for the relationships between the compiler, linker, and environment, that suggest the form of a solution. In section 3.6 I will describe a simple bootstrap loading facility based on this model, which treats the host machine as one of the execution environments of the operating system. This concept will allow elaboration of intricate initialization clauses for permanent objects. In chapter 5 I will use monitored types in the host environment to automate the configuration of environment management sets, such as process sets and page tables.

3.4.5. Summary

In this section we have identified four interesting classes of type management: value, object, monitored, and allocated types. We have seen how coupling may be manifested in each, and how coupling constrains the ways in which type instances may be dispersed among several environments. We have seen that monitored types could have been widely used in FAMOS, yet are difficult to program satisfactorily when the type instances are permanent. This is often the case for virtual resource types such as segments, processes, and software trap vectors, when used within a multi-layer operating system.

In the next section we will develop a conceptual framework and notation for binding program units to environments. We will judge its suitability by seeing how well it supports the type management classes identified above. In particular, we shall look to see how monitored types can be supported in multiple environments having overlapping logical memories.

3.5. A Notation For Environment Bindings

We have now laid all the groundwork needed to develop and assess a notation for programming the connections between program units and environments. We have developed a way of representing an environment specification as a source language module. We have identified four classes of type management that we would like to support. We have discussed examples of environments from several operating systems, from which we can develop realistic protection scenarios.

The notation we develop should satisfy the following criteria:

- Fitness: it should let the system designer specify bindings between environments and source language program units, rather than object modules. The designer should have complete control over the use of resources to support programs.
- Clarity: the notation should facilitate reasoning about the use of environments in systems.
- Brevity: binding information should not clutter programs unnecessarily.
- Flexibility: the notation should support the type management techniques and protection scenarios discussed earlier.
- Modularity: binding information should not corrupt the modularity of the system.
- Implementability: the notation should lead to a straightforward implementation.

First we shall define a notation for types and modules, which emphasizes the distinction between an abstract definition and a concrete implementation. We shall

discuss the appearance of these concepts in existing languages, but retain our own notation for clarity.

Next we add to the module syntax the mechanism for environment bindings. Each kind of program unit can be viewed as a type implemented by the compiler. An environment name attached to a program unit declaration becomes a generic parameter to that instance of the type. The compiler uses the environment named to implement the program unit. Binding a source-language variable to a module implies binding all of its primitive components to that environment. Binding a source-language procedure only *constrains* the procedure to be compiled for the named environment, *if* and *when* it is instantiated. The translation system is permitted the freedom to instantiate a procedure separately, expand it in line at its call sites, or eliminate it altogether if it is not used. Binding a type or a module to an environment binds all of its component variables and procedures to that environment, but not inner types.

In contrast to an ordinary procedure, binding an entry point procedure to an environment normally causes instantiation of the procedure, since it may not be known until runtime whether the procedure will be invoked, and it cannot be expanded in line in some other environment. This in turn forces instantiation of all the procedures it calls. In addition, the environment module records the entry point for integration with the protected procedure call mechanism. Parameters to entry point procedures can be by-reference, in operating systems that support segment sharing between environments.

Next, we briefly discuss how to control access to an environment manager, i.e. how to demarcate the set of system components that may may place code and data in an environment. Because an environment has a source language name, ordinary scope mechanisms are sufficient. We see how to describe FAMOS multi-level environments using an **acquires** clause to limit the scope of the environment name.

To see how the notation thus defined works in practice, we apply it to the type management techniques identified in section 3.4. We see that binding information can be added to each of them conveniently. In particular, we see an example of using an entrypoint procedure in the implementation of a monitored type without making it visible outside the type management module. To gain further experience with the notation, we reprogram FAMOS semaphores.

3.5.1. Definition vs. Declaration

Before we can develop a technique for binding program components to execution environments, we need to understand the relationships between an abstract *definition*, which might be independent of any environment, and a concrete *declaration*, which must be attached to one or several environments.

When we write a type *definition*, we give the structure of the object (in high-level terms), and a set of *operations* on the object (written as type-safe algorithms). The type definition itself is independent of the use to which the type will be put. As soon as we *declare* an object of that type, however, we are faced with questions:

- Where will the object be located?
- How will the operations be implemented?
- What other objects must the operations be able to access?
- Which other objects will be coupled with the new object, and how?

In single-environment programming systems, all of these decisions can be left to the compiler, which can do either something reasonable or something optimal. However, to transform existing programming styles into ones which accommodate multiple environments, we must come to grips with the relationship between a type definition as an abstract entity and a type manager as a body of executable code.

The terms *module* and *type* have acquired very similar meanings in modern data abstraction methodology. Each denotes a program component that can restrict access to its constituent parts: a type restricts access to its component fields; a module restricts access to its component variables and other elements. If there is a difference, it is more one of nuance than substance: In some languages, a type definition describes a data structure with associated operations, which can be instantiated many times, whereas a module is instantiated only once.

Although most of the examples I will write could be programmed in existing data abstraction languages, I wish to emphasize the distinction between modules and types, and to suppress many of the details of real languages. Therefore, I shall use the following definitions and notation:

- A type definition is a group of variable, operation, and type definitions. Each of the variable definitions (formally identical to variable declarations) defines a component of the representation of the type.
- A variable declaration names the type of which the variable is an instance. Elaborating the declaration creates an instance of each of the component variables of the type.
- A procedure defined in a type is privileged only in the sense that it can use the names of components of the type. The first parameter to a procedure may be written as a prefix parameter, to differentiate identical procedure names defined on different types. However, if a procedure takes additional arguments of the same type, it may access the representation of each of them equally well, using the component names.
- A name defined within a type and exported from it will normally be referred to only as a component of an *instance* of the type, and not a component of the type itself.
- Qualified names may be abbreviated wherever doing so does not cause ambiguity.

- A module is a shorthand way of combining a type definition with its only instance.

```
type A is
   provides D, F
   var B,C:integer
   proc D ( X:A, Y:A, Z:boolean )
        if Z then X.B := Y.C        -- full qualification

   proc F ( X:A, Z:boolean )
        if Z then B := 0            -- abbreviation

        type G is . . .

end type A

var H,I: A
var J: H.G        -- Type I.G is a different type from H.G
                  -- A.G would be illegal

module B is . . . <<type body>> . . .
```

Figure 3-13: Skeletal Program Showing Types

Figure 3-13 gives a schematic program which illustrates the notation. The module B defined at the end is a shorthand for

```
type UniqueName is -- type body
var B: UniqueName
```

Such a module definition would contain a group of variables, procedures, and types, with import and export clauses, just as one would expect of a type. It is merely a combination of a type definition and a declaration for the only variable of that type.

A type definition has no built-in facility for declaring variables that are to be shared among all type instances. That affect would be achieved by enclosing the type definition in a module or another type definition. Consider the following:

```
module A
   provides B
   var C

        type B is
             requires C
             provides ... << operations >> ...
              . . . << type body >> . . .
        end type B

end module A
```

Since module A has only a single instance, there is only one instance of A.C, which is visible within the implementation of type B, making it a shared variable. The reason for not allowing shared variables within types, is that the shared variable represents a central type manager. The structure above makes the manager's existence more apparent.

Sometimes one wishes several structurally identical yet distinct type managers, such as storage managers. This can be achieved by replacing the word "module" above by the word "type", and declaring several instances:

```
Type A is
   provides B
   var C
        type B is
             requires C
             provides ... << operations >> ...
               . . . << type body >> . . .
        end type B
end module A
var X,Y,Z: A
```

There are three distinct type managers, X, Y, and Z, managing the types X.B, Y.B, and Z.B. Each distinct type has a copy of C to share among its instances.

The definitions I have chosen for *type* and *module* will generally allow uncoupled types to be described simply as type definitions. However, to describe a type whose instances are coupled by shared data, one must imbed the type definition in an enclosing module or type, so that there will be an explicit instantiation of the shared data, and so that the type name is qualified with the name of that module or type instance, which then becomes the type manager.

The concepts defined above appear in various combinations in current languages. Let us examine four: Modula, Euclid, Ada, and Alphard.

In Modula, a module description is simply a collection of procedure and variable declarations, with a boundary drawn around it to limit the visibility of names. A module *may* also contain type definitions, which are non-forgeable templates for declaring structured variables. There are no explicitly designated operations on such types; there are only procedures declared in the same module as the type definition, which may use it to access the representations of their parameters. A Modula module, therefore, defines a centralized type manager for the types within. Although the type instances themselves may be located anywhere, they may be operated on only by passing them to one of the explicitly-declared procedures.

Euclid modules are patterned after Modula modules, but with one important exception: a module description may be used as the text of a type definition. Therefore, there can be any number of instances of a module. If the module description includes a variable declaration, then each instantiation of the module defines a distinct instantiation of the variable. If the module description includes a type definition, then each instantiation of the module defines a distinct type. Consequently, each time that type name is used, it must be qualified with the name of the module instance which is to provide its implementation. Similarly, to invoke a

procedure defined in a module, one must name the module instance which supports the procedure. There could be a separate procedure implementation for each module instance, or the module name could be viewed as a "prefix parameter" to the procedure. Euclid has explicitly allowed the translator this freedom; it goes even further in stating that the notation **inline** is considered non-binding advice to the compiler. Since a type management module in Euclid can itself be the text of a type definition, there could be several different type managers for textually identical types. This would be highly desirable if each type manager maintained a resource pool for use by its instances.

An Ada **package** is a module containing constant, variable, type, procedure, and package definitions. It is the type manager for any types it exports. Procedures may be marked **inline**, so that the type manager need not be viewed as entirely centralized. Generic packages offer some flexibility for creating multiple instances of a type manger. However, it is not possible to define a type whose instances are type managers, nor is it possible to define a type, one of whose components is another type.

Alphard makes no distinction between a module and a type; both are written as **forms**. If several instances of a form are to be monitored by a central type manager, that manager is usually written as an enclosing form, so that each reference to the inner form must specify the outer form instance which will manage it. See figure 3-14. Alphard allows a form to declare a variable to be shared by *all instances of the form*. This hidden variable would have to be accessible by every instance of an operation on the form; thus, it forces centralized implementation of the type manager without saying so in the specification.

```
form Set is spec
     form Element
     function Contains ( E:element )
end spec
impl Set is
     impl element is
          var data
          var link:ref element
     end impl element

     var Inlist : ref element

     body Contains is
          Temp := Inlist
          while Temp not equal NIL do
               if Temp equal E then return TRUE
               Temp := Temp.next
               end
          return FALSE

  . . .

end impl SET
```

Figure 3-14: Monitored Forms (Alphard)

In the notation I have defined, creating a module, or an instance of a type, involves creating only instances of the representation, and not necessarily the operations. They might be marked as **inline**, or they might not be instantiated if they are not invoked. This is particularly significant if the type comes from some library of type definitions. One would hope that creating an instance of a complex number, for example, would not force creation of procedures for all the operations which might have been defined with it in the library.

We will focus our discussion of binding on the relationship of variables to environments. We must eventually ask, however, when it is that the operations do become translated into real machine-language instructions.

3.5.2. Binding Program Units to Environments

We said in section 3.3.4 that environment information could be specified to a compiler as a generic parameter to procedure and variable declarations. We shall now integrate this notion into our notations for modules and types. We start by reviewing the mechanisms by which a compiler places machine language objects into environments. Then, we consider each kind of source language object in turn, from the simple to the complex: variables, procedures, modules, and types.

In the examples which follow, we use <pointed brackets> to indicate generic parameters that are environment bindings, recognizing that the syntax is poor, but explicit. Remember, variables and procedures are themselves types; environment names are parameters to instances of these types. The pointed brackets merely highlight those parameters which happen to be environments.

3.5.2.1. Primitive Machine Language Objects

A compiler represents all program units as primitive code and data objects, as supported by the environment for which it is compiling. A data object depends on the environment to provide the storage in which it resides and to define the instructions that implement the operations of its type. A code object depends on the environment to provide storage, an instruction set, an instruction interpreter, and a virtual memory mechanism. A data object may be accessible from several environments as long as they all support its primitive type, but normally a code object may only be invoked from within the environment in which it resides, because the virtual addresses it uses depend on the particular address translation database of that environment. A compiler is free to concatenate several primitive objects into *a section*, if they are to reside in the same segment. The environment module provides:

- the program region, with operations to allocate space for sections
- the virtual memory manager, with whatever operations the compiler might need to determine virtual addresses

- the name of the architecture description the compiler should use to generate machine code

3.5.2.2. Variables

Assuming that an environment E is visible in a given scope, we would write a variable declaration as

var<E> X: **integer**

Recall that **var** is a type implemented by the compiler; the environment parameter provides the program region segment, and primitive type **integer**, out of which to build the **var**.

A user-defined type would be instantiated in a similar fashion. Consider the declaration

var<E> Y: **usertype**

Assuming that **usertype** is, say, a value type, like **complex**, each primitive component of its representation would be instantiated in environment E.

3.5.2.3. Procedures

The syntax for binding a procedure to an environment is analogous to that for variables. Consider

proc<E> Y (A,B:**anytype**)

However, binding a procedure to an environment does not necessarily imply that the procedure will be instantiated. Depending on the cleverness of the translator, the procedure might be instantiated once, many times, or not at all. I take the position that binding a procedure to an environment *constrains* that procedure to be instantiated only within that environment. A simple compiler would instantiate such a procedure, exactly once. A language supporting **inline** procedures, such as Euclid, could expand them at any call site within the specified environment. A more clever compiler would choose whether the procedure were inline or out-of-line.

One might argue that the translator should be given even more latitude. The environment binding might indicate the minimum requirements of the instantiation environment. The translator might be allowed to instantiate the procedure in any environment whose logical memory contained that of the named environment, whose virtual memory was implemented by the same address translation database. However, this possibility is quite speculative, so I will save it for future investigation. Instead, I simply allow a procedure to be explicitly marked **free**, meaning that it may be instantiated in any environment that can access the objects it manipulates.

3.5.2.4. Types and Modules

Attaching an environment name to a *type definition* attaches the environment name to each constituent variable and procedure of the type. Instances of the type can only be declared in *source language* modules in which the environment name is visible, and are bound to that environment. The operations on those instances are constrained to execute in the same environment, unless they are marked **free**. In either case, the operations need not be instantiated unless they are used.

Attaching an environment name to a *module declaration* is a shorthand for attaching it to both the variable and the type embodied by the module. Thus, all of the constituent variables and procedures named in the module would be bound to that environment, as would *their* constituents. Whether or not the procedures in the module are instantiated depends on whether or not they are used.

Binding to a type or module is transitive to inner modules (because they are constituent variables), but not to inner types. If an inner type is not exported, then all of its instances will be bound anyway, being components of the outer module. If the type *is* exported, binding it to the environment of the enclosing module would be too restrictive. If the type is to be both bound and exported, the module would import the environment name and bind the type explicitly.

Although binding a type or module to an environment constrains the procedures defined within, binding an *instance* of a type does not constrain the operations on *that instance*. Because there might be other instances of the type in other environments, and because the type might define binary operations on instances, we must allow each instance to be operated on from any environment that can access it.

The rules I have chosen above may appear to be unduly irregular. Certainly I could have chosen to make all bindings transitive to all inner units, using the **free** attribute to override this when necessary. Instead I have chosen the "defaults" that I believe will fit the most common usage. In addition to attaching an environment name to a declared object, one can pass environments as ordinary declaration parameters, or place them in **provides**, **requires** and **acquires** clauses. Thus when one desires to bind only certain components of a module or type, he may import the environment or make it a generic parameter, and specify exactly which components are bound.

Observe that the rules above make it possible to place an ordinary self-contained program, of any size or complexity, into a single environment, by a single annotation. The program is a module, so binding that outer module to an environment binds the representations of all variables and procedures in the module, at any nesting depth. Types defined within the module would be bound according to how they were used: the variables of that type, occurring within the module, would be bound by the top-level binding. If the module exports a type, then any type instances occurring outside the bound module would not be bound.

3.5.2.5. An example: FAMOS semaphores

Although we have not yet discussed how entry points provide inter-environment communication in the proposed notation, we have sufficient tools to describe the static structure of a multi-environment type.

We referred in Section 2.1.2 to FAMOS semaphores, whose representations spanned two environments. We could program the representation of such semaphores as in figure 3-15. The semaphore S would have its WaitCount in the environment E, yet its WaitingList would be in the process environment. If the semaphore were a permanent object there would never be a run time computation of the address of S.w based on the address of S. Instead, wherever S.w appeared, the linker would substitute the address, in the process environment, of the waiting list.

```
type WaitingList is
     requires ProcessEnv
     provides Insert, Remove
     var<ProcessEnv> WL:list[ProcessDescriptor]
       . . .

end WaitingList

type Semaphore is
     requires ProcessEnv
     var WaitCount: integer := -1
     var W: WaitingList
       . . .

end Semaphore

var<E> S: Semaphore
```

Figure 3-15: FAMOS Semaphores, With Environments

In section 3.5.4 we will expand this example with the operations on semaphores and waiting lists, showing how the entrypoint procedures connect the environments.

3.5.2.6. Protected Procedures

We have defined an *entry point*, in Section 3.1.4, as a designated logical address in the program region of an environment, to which control may be transferred from some other environment. In Multics, Hydra, and FAMOS the entry points of *address spaces* were used to implement *protected procedures*, so that a module could obtain runtime address protection for its data structures. To incorporate such runtime facilities into the implementation language, we must analyze its role in the relationship between source modules, environments, and protection domains.

A procedure call can cause any of the following transitions:

- between modules
- between environments

- between protection domains

Since two modules might reside in the same environment, not every module interface procedure should be an entry point. Since two environments might coexist in the same protection domain, not every entry point to an environment will cause a change of address space. Since a single module might occupy several interacting environments, entry points should be concealable.

Therefor, we adopt (and adapt) the technique used for Hydra and FAMOS, namely, providing in the implementation language a way to explicitly mark the procedures that are points of entry to environments. In Bliss-11 the denotation was used only to generate the proper procedure linkages; we shall also use it to record the entry point in the environment description. The syntax is straightforward:

entryproc<E> P(. . .)

This gives the author of the procedure the right to designate it as invocable from other environments. Anywhere an **entryproc** can be named, it can be invoked. Compiling such a call would entail retrieving from E the necessary protocol, and possibly also place in the invoking environment's representation a *capability* for P.

With entryprocs so defined, we can now restrict the use of non-entry procedures:

> A procedure bound to an environment may be invoked only by other procedures bound to the same environment.

Thus, control may flow from one environment to another *only* via **entryprocs**.

Since **entryprocs** are to be explicitly-named entities in source programs, they can concealed within modules. We will shortly see an example where a module provides ordinary, unbound procedures to its users, each of which contains a concealed call to an **entryproc** of the module's protected environment. The module could even conceal the very existence of its private environment.

Parameters to entrypoint procedures require special support from both the operating system and the language system. The parameter mechanism support will often involve the address translation facility and the overall protection mechanism of the operating system. Even if the operating system can support by-reference parameters between environments, the translation system might still have to specify when to make the actual parameters addressable.

Segmented virtual memories combined with capability systems, such as in Multics and Hydra, make possible by-reference parameters to entryprocs, between otherwise suspicious environments. Each protected procedure refers to its formal parameters by local names reserved for them. The protected procedure call mechanism sets up the local name translation table entry to refer to the actual parameter segment. VAX/VMS, in contrast, provides protected procedure calls only for entering protection rings. In that system, the caller's address space is always a subset of the environment it is entering, so by-reference parameters are

implemented simply by passing addresses. In general, one would expect system designs exhibiting some mixture of these two approaches. Sometimes the caller's and callee's address spaces will be disjoint; sometimes they will intersect; sometimes one will contain the other; and sometimes they will be identical. Depending on the degree of overlap between environments, an appropriate protocol can be found (perhaps with the help of a **pragma**), to bring about the transition between environments without undue overhead.

3.5.3. Limiting Access to Environments

By making environments explicit objects in system descriptions, we have made it possible for a single module to declare objects in several environments. However, in so doing we have given up the right to create a single list of all the objects in a given segment. We did so willingly, but we still need to be able to *limit* the set of objects in an environment, so that only "authorized" modules have access to it.

Scope limitation facilities in modern languages provide a perfectly reasonable means for doing this. A module defining an environment provides a name denoting the right to create objects in it. (Typically, this would be the allocation procedure.) A system designer would limit the right to place objects in an environment by limiting the scope of that name. He would define the modules allowed to place sections in a given environment by surrounding them with a module which did not export the name of its allocation procedure. In many cases a suitable module would already be present, delimiting a virtual machine. In other cases, more selective access control is required.

Sometimes a system designer might wish to protect a module from *most* of its users, by placing it in a protected addressing environment, but still allow the possibility of adding more procedures to the environment from within a higher level module. This arrangement occurred frequently in FAMOS [Habermann 76] The specification of the protected module cannot regulate what modules would have access to its program region; that would imply that the correctness of the module depended on how it was used. Instead, the higher-level module which was to fill up the environment would **acquire** it, as in figure 3-16.

```
module BaseMachine is
   provides BaseEnv, OtherThings
   . . .
end module BaseMachine
```

```
module Feature1 is
   requires OtherThings -- but not BaseEnv
   . . .
end module Feature1
```

```
module Feature2 is
   requires OtherThings -- but not BaseEnv
   . . .
end module Feature2
```

```
module MoreBaseMachine
   acquires BaseMachine.ProgramRegion
   . . .
end module MoreBaseMachine
```

Figure 3-16: Selective Access to a Program Region

If MoreBaseMachine were not present, Feature1 and Feature2 could both require BaseMachine.ProgramRegion. This is perfectly proper, since MoreBaseMachine is the only module whose correctness depends on keeping Feature1 and Feature2 out of the BaseMachine environment.

3.5.4. Binding Type Managers to Environments

Our notation for environment bindings is now essentially complete. To gain familiarity with it, and explore its utility, let us look at how various classes of type managers (identified in section 3.4) would be programmed using these binding notations.

A value type would be written as a pure, unbound type, probably with all of its operations marked inline. Any particular instance could be bound to an environment. The compiler would have the freedom to instantiate the operations in any environment that can access the instances. Values could be freely passed from one environment to another, as value or copy parameters. Each formal parameter to an entryproc could be a separate instance of the type.

Uncoupled object types would be treated in the same way as value types, except that they cannot be passed as value parameters. Each operation on the instance would have to occur within an environment which could address its representation. Binary operations pose certain problems, however, as in figure 3-17.

```
type T
     provides BinOp
     proc BinOp( X,Y:T )
       . . .
end type T

var<E> X: T
var<F> Y: T
. . .
X.BinOp( Y )
 . . .
```

Figure 3-17: Binary Operations on an Object Type

The invocation "X.BinOp(Y)" would have to occur in an environment which could address the program regions of both E and F. This could happen if one environment's logical address space contained the program region of the other. Another way of handling it would be to pass Y as a by-reference parameter to a protected procedure provided by E; this would bring Y into E for the duration of the call, and thereby enable X.BinOp to do its work.

A coupled object type could be programmed as in figure 3-18. The Bank has a central cash supply through which all monetary transactions must flow. The Bank allows its customers (users) to examine balances, but not change them directly. Instead, the customer must present the account to the bank for each transfer. The central fund and transfer procedure are both bound to environment E, but individual accounts may be created in other environments. A customer may examine his account without entering environment E, but must go there to transfer money between the central fund and his account. Conversely, the bank cannot move money out of an account unless the customer calls *transfer*.

```
module<E> Bank
    provides Account [with Examine], Transfer

        type Account is
            provides Examine, Balance

            var Balance: dollars
            proc Examine ( A:account ) : dollars =
                return A.Balance
            end type account

    var CentralFund: dollars

    entryproc<E> Transfer ( A:account, S:dollars ) : ErrCode =
        if S >= - A.Balance then begin
            CentralFund := CentralFund - S
            A.Balance := A.Balance + S
            return success
            end
        else return failure
        end transfer
    end module Bank
```

Figure 3-18: Coupled Object Type

For an example of programming monitored types, we rewrite the clock manager of figure 3-12 as in figure 3-19.

```
module<ClockEnv> ClockManager is
    provides Virtual Clock

    type Clock Data = . . .
    var Count := 0
    var ClockTable: flex vector [1 .. Count] of Clock Data

        type Virtual Clock (StartTime) =
            requires Count, ClockTable
            provides Start, Stop, Reset
            const cd: ref ClockArray[]

            entryproc<ClockEnv> Start ( C: Virtual Clock ) = . . .

            entryproc<ClockEnv> Stop  ( C: Virtual Clock ) = . . .

            entryproc<ClockEnv> Reset ( C: Virtual Clock ) = . . .

            entryproc<ClockEnv> Create returns ref ClockTable[] =
                HighExtend ( ClockTable )
                Count := Count + 1
                return ref ClockTable [ Count ]
                end

            initially begin
                cd := Create ( )
                Reset ( self, StartTime )
                end

            end Virtual Clock
end Clock Manager
```

Figure 3-19: Multiple Environment Virtual Clocks

ClockTable is a component of the module "ClockManager", so it will be attached to ClockEnv. The type "VirtualClock" is *not* bound to "ClockEnv"; instances can be created anywhere. A "VirtualClock" obtains a pointer to an element of "ClockTable" during initialization, by invoking the **entryproc** "Create". Recall that a constant is a variable whose value doesn't change *after initialization*.

Observe that although "Create" is an **entryproc**, it is not *visible* in the source program outside the type definition for "VirtualClock". Therefore, it can be used only as described there, namely, to obtain a table entry for a VirtualClock.

This "invisible" entryproc satisfies a major goal of modular programming: it distinguishes the structure of the executable representation from the structure of the source program [Parnas 71]. The entryproc is unquestionably a feature of the run-time interface between the clock environment and other environments; yet, because the compiler can invoke the environment manager to create the entry point, its existence is known (in the source language description) only to the type manager for virtual clocks. Should that type be reimplemented, changing the specifications for the entryproc, no other system components would be affected.

The Virtual Clocks thus created are now implemented such that each clock user is oblivious to how many other clocks there are; the right to request operations on a clock descriptor is protected by the source language type mechanism; the clock descriptors are protected by the addressing mechanism; the number of clocks is determined by demand, rather than by fiat; and, all of the clock descriptors are continuously addressable by the clock management module.

Type managers for allocated types require no further programming innovations. The objects themselves are private to the managing module, and created in the corresponding environment. The manager must define and export an unbound type which can hold a *capability* for a resource. If the resource is pre-emptible, the holding type must be monitored. If the resource is dynamically allocated, but not pre-emptible, then the holding type need only be coupled to the management data structures. If the resource is allocated statically, the binding type may be a value type.

Each variable in each example above was associated with just one environment. Some *modules* contained elements bound to different environments, but each *type* kept all of its immediate representation in a single environment. Let us reexamine FAMOS semaphores once more, to see the implications of declaring an unbound procedure which operates on a multi-environment object. In figure 3-20 I have reproduced figure 3-15, with the addition of the procedures *WaitingList.Insert*, *Pause*, and *Semaphore.P*

```
type WaitingList is
     requires ProcessEnv
     provides Insert, Remove
     var<ProcessEnv> WL:list[ProcessDescriptor]

     proc Insert ( W: WaitingList, P: ProcessDescriptor ) =
          requires ProcessEnv.ProgramRegion
               -- the usual list insertion text has been omitted

       . . .

end WaitingList

var<ProcessEnv> ReadyQueue: Queue of ProcessDescriptor
var<ProcessEnv> CurrentProc: ref ProcessDescriptor

entryproc<ProcessEnv> Pause (W: WaitingList) =
          Suspend (CurrentProc)
          Insert ( W, CurrentProc )
          CurrentProc := Remove(ReadyQueue)
          Continue (CurrentProc)

type Semaphore is
     requires ProcessEnv.ProgramRegion
     var WaitCount: integer := -1
     var W: WaitingList

     proc P ( S: semaphore ) =
          If (Waitcount := Waitcount + 1) > 0
                    -- indivisible increment and test
               then Pause ( W )

     proc V ( S: semaphore ) =
       . . .

end Semaphore

var<E> S: Semaphor
```

Figure 3-20: FAMOS Semaphores, With Procedures Added

The procedure *Insert* is not bound to any environment, but its text refers to WaitingLists, so it requires the program region to which they are bound. In a language where a procedure is an open scope, the need for that program region would have to be derived from the text.

The procedure *Pause* is bound to ProcessEnv, so it is assured of access to both the waiting lists and the ready queue.

The procedure *P* passes its waitinglist to *Pause*, without ever addressing it. However, imagine the situation if Pause were an unbound procedure, rather than an entryproc. The compiler would deduce that, since WaitingList.Insert needed to address the waitinglist, so would Pause, and so would P. Therefore, semaphore operations would only be allowed in environments which could address ProcessEnv.ProgramRegion.

These two versions of the semaphore module illustrate two ways of using multi-

environment objects: passing the components explicitly to the appropriate environments, and deducing legal operations by induction on parameter chains. Either way, the notation gives enough information to determine from the source text whether the desired operation is feasible and accomplishes the intended deed.

3.5.5. Summary

We have defined a notation for types and modules, and for describing the associations between program elements and execution environments. Let us consider how the notation measures up to the criteria we stated for it:

- Fitness: The system designer can attach an environment binding to any piece of a program, large or small.
- Clarity: By embedding environment bindings in the source text, one can relate that information to other aspects of the system structure.
- Brevity: The transitivity of bindings allows simple systems to use few binding clauses.
- Flexibility: The notation supports unbound types, types with bound representations but free operations, and bound types. Each of the four classes of type management identified earlier can be programmed conveniently in a multi-environment domain. We were also able to reprogram parts of FAMOS conveniently.
- Modularity: Access to entrypoints and access to program creation facilities can be controlled with the same scope mechanisms as other source language entities, without violating the source language modularity of the system.
- Implementability: Each environment binding directs the compiler to the program implementation resources it needs, in a straightforward fashion. Section 7.4 contains a detailed discussion of implementation issues.

3.6. Comprehensive System Descriptions

In this section I will show how to use environment bindings to integrate all the pieces of a multi-level, multi-environment system into a single, comprehensive system description. The notation defined in the previous section allows us to describe the *structural* relationships between system components. In this section we will detail the structure of a typical cross-compiled system, and describe the *control flow* between levels during translation, initialization, system generation, loading, and startup. When we can trace that control flow from the beginning of translation to the end of startup, we will have achieved a comprehensive system description.

We will address two interrelated problems: how to include host-machine activities in the overall system design, and how to sort out the initial bookkeeping activities at each system level so that they can be correctly sequenced.

I shall incorporate the host activities by treating the host as one of the environments comprising the system. The lowest system level is a module containing both host and target components, which communicate with one another via the medium used for loading. Typically the host component is a "down-loader" that writes a segment to, say, a tape. The target component is a *bootstrap loader*, which reads the segment from the tape. Each subsequent system level may bind components to both host and target environments. These components communicate via whatever facilities are provided by lower levels.

I shall differentiate three classes of bookkeeping tasks that take place during the instantiation of a system: initialization, system generation, and startup. The initialization activities of a level take place *before* it is used to implement subsequent levels, and consist primarily of putting the data structures of the level into well defined initial states. System generation actions take place during and after instantiation of subsequent levels, but before the system is transmitted to the target machine. Startup activities are sequenced "bottom to top" on the target machine. Each system level receives control from the underlying virtual machine, activates its own facilities, and passes control on to the next higher level.

A system description is elaborated in order from lowest to highest virtual machine level. Each module, in whatever environment, may specify an initialization clause that is to be elaborated when that module is instantiated. The system generation activities of a given level are written as host-environment procedures, which can be invoked during elaboration of subsequent virtual machine levels, either directly from initialization clauses, or by the translator when, say, instantiating a monitored type. The startup activities of a level are written as target procedures invoked by a *handler* for the condition *startup*.

I present the problems and their solutions in the context of an extended example: a small loader for a PDP-11. The system has four execution environments: the host environment, the bootstrapping environment, the loading environment, and the user environment. I present partial descriptions of each module and environment, with commentary on the conceptual relationships between components and on the details of key system interactions. Assembling the module descriptions thus presented gives the skeleton of a comprehensive system description, with enough detail to understand the flow of control from the beginning of translation to the end of startup.

3.6.1. A Simple Loader

The example I have chosen to describe is just about the smallest "operating system" one could conceive: a loader. It is a "toy problem" in the sense that one can understand how a loader works without the environment concepts and notations developed in this thesis. However, even such a small system contains several

distinct execution environments, has components on both host and target machines, and requires careful sequencing of initialization, generation, and startup. I have purposely chosen a small example so that the reader may concentrate on the *representation* of the solution without first having to grasp the design of a complicated system, and without having to wade through a great deal of detail. In chapter 5 we will examine the system instantiation problems of a more substantial system, to assess the power of the proposed methodology in more realistic situations.

The system we will describe is a very conventional loader for a PDP-11. It consists of a bootstrap loader residing in block 0 of a DecTape, which loads and starts whatever program starts in block 1 of that tape. That program is itself a loader, which loads and runs the user program residing later on the tape. This second loader might be there for the purpose of allowing the user to choose interactively among several programs on the tape, to set up debugging aids, and so on. Problems faced here include: transmitting information about the size of the second loader and the user program; coordinating tape usage among several system levels; coordinating use of primary memory during loading; and formalizing the sequence of bootstrapping operations taking place on the target machine.

3.6.2. The Host Environment

Conventional system instantiation technology operates as a series of passes over the system description. A compiler translates the source files into a set of object files. A linker translates the object files and linker command files into a set of segments, again in files. A system generation program connects the segments, constructing segment descriptors and process descriptors, coordinating linking and relocation, allocating memory for system components, and so on. A separate down-loading program transfers the entire system to some medium suitable for loading. On the target machine, a chain of bootstrap loaders brings the system into memory and sets it in motion.

Assembling objects into segments and constructing environment management data structures are as much a part of an operating system as scheduling processes and handling page faults. In order to include these activities in system descriptions, we shall include the host environment as one of the environments in which the operating system resides. This allows us to declare system generation procedures just like other procedures, the only difference being the particular environment they are bound to.

Because they execute in different environments and are usually written in different languages, host and target machine portions of an operating system have customarily been kept in completely separate modules, which communicate with each other via the loading medium. Since the programs that generate a system and the programs

that load it share information about a great many design decisions, they ought to be combined into modules that conceal the shared information. We would like to transform the multi-pass view of system instantiation into one in which the host-machine representation for each program component resides in a host data structure provided by the environment manager responsible for it. Each environment manager would provide host-machine procedures to enter program representation information into the structures, and to communicate the contents of those structures to the target machine.

These host data structures would use the primitive type **segment** to contain the representations of target-machine program units. The linker would appear in a system description as a host-machine module providing the abstract type **segment**, and a host-machine representation for them. One possible representation would be the name of the host file containing the segment, together with some indication of whether the segment has been linked yet. With **segments** as atomic data items, the operating system can include host programs which link a segment, define its relocation base, examine its length, and copy it to a tape, disk, or communication line. Such programs could also build tables of segment descriptors, create directories, allocate storage, lay out page tables, and so on.

The specification for a linker would look something like the following:

```
module<host> Linker is
   provides Segment, Section, CoreBlock, LogicalNameTable, LogicalMemory

        type<host> segment is
           requires Section
           provides Insert, Link, Copy

           path Insert* ; Link ; Copy* end

           proc Insert ( Seg: segment, Sec: section) = . . .
           proc Link ( Seg: segment ) = . . .
           proc Copy ( Seg: segment, TB: tape block ) = . . .
           . . .
        end type segment

   type<host> Section is . . .
   type<host> CoreBlock is . . .
   module LogicalNameTable is . . .
              -- Maintains global symbol table
   type<host> LogicalMemory is . . .
              -- a set of segments accessible to an environment
end module Linker
```

The type **segment** actually takes several optional parameters specifying size, base address, and so on. One of them is the core block in which the segment is to reside. The linker can check whether two or more segments residing in the same core block will overlap, or it can compute the base address of one from the limiting address of another. The type **section** is the unit of program representation that the compiler can place in a segment, via the procedure **Insert**. All sections

must be inserted before linking takes place. After linking, the system generation program can make as many copies of the segment as it needs. The logical name table maps each target object identifier in a system description to an ordered pair <segment, displacement>. It conceals the problem of unresolved references, maintaining whatever "fixup lists" are needed. The type LogicalMemory will be used by environment modules to record the logical segments addressable in a particular environment. It provides operations to test whether a given symbolic address is defined in any of the member segments, so that the language system can determine if a procedure compiled for a given environment will be able to access the objects it needs.

3.6.3. Initialization vs. Generation vs. Startup

In section 2.2.2 we described the many sorts of problems which arise during system integration, often due to lack of coordination between initialization, generation, and startup.

At each system level one can usually separate initialization actions from generation actions, and those from startup actions. However, one cannot simply initialize all levels, then generate a system, then start it up. If each system level is going to participate in the representation of higher levels, then at least the host machine portions of a level must be fully activated before the next level can be translated. Conversely, if a system level is going to contain static data structures configured to the needs of the program it is managing, then the target machine portion of a system level cannot be generated until the program it is managing has been compiled. Part of the problem comes from the distinction between "compile time" and "run time". If we insist on compiling an entire system, then running it, we will not be able to describe the system generation process.

In single-environment compilation technology, it is no longer necessary to distinguish "compile time" from "run time". One speaks simply of elaborating a program. Each module of a system may contain initialization code, which is executed immediately after the module is created. Assuming the module is permanent, it is initialized before any subsequent module is created. The initialization code for the outermost module is none other than the "main program".

If one were to apply this paradigm to cross-compilation technology, one would be tempted to view the initialization code for each module as serving the needs both of initialization and startup: any actions that a virtual machine must perform before beginning to execute the programs it is managing, would be stated in the initialization section. However, this would not work, because some actions must take place on the host machine, and some on the startup machine, at different times:

- Some startup actions require interaction with the hardware and permanent data on the target machine. The translator could not

simulate those actions on the host machine, and would therefore have to put off all initialization until after the system was compiled, generated, and transported to the target machine.

- Even if startup could be accomplished on the host machine, requiring all modules at a given level to be started before translating the next level, would preclude customizing the static structures of a level to the program it is managing.

Instead, we shall separate initialization from startup, in such a way that initialization can be performed during compilation, and startup deferred until after loading.

First, we observe that the concept of a "main program" simply does not apply to an operating system. A running system does not normally have a master procedure that carries out the task of running a machine. Instead it is simply a set of facilities, which are invoked by user programs, or which respond to external events.[5] Bootstrap loading can be viewed as a chain of responses to an event, namely someone pressing the "start" switch on the console. This is analogous to the way existing systems typically recover from a crash. A trap routine fetches a simple loader from some secondary storage location. That loader dumps the core image into an error log, then patches the system back together and restarts it.

We shall write each bootstrap procedure as an exception handler for the condition "startup", raised by the underlying virtual machine. This allows each system level, from the bottom up, to obtain control of the cpu for the purpose of setting itself in motion, loading the program it is supposed to support, then signalling "startup" to that program.

The exception handling I have in mind follows the work of Levin, in that one can *declare* an exceptional condition, provide a *handler* for a condition, or *raise* a condition (signal that the condition has arisen) [Levin 77]. A condition may be *signalled only from within the scope in which it is declared*. At this level we allow only one handler for each condition, and require that handler to be permanently enabled for handling the condition.

A condition must be associated with an environment, at least insofar as the representation of the condition handler register must be placed somewhere. I adopt the convention that *placing a condition in an environment constrains its handler to be invocable from that environment*.

With startup thus taken care of, the initialization clauses of modules can be used exclusively for actions associated with module instantiation. Elaborating a program takes place in the order in which modules appear in a system description. As the translator instantiates each permanent module, it carries out the specified initialization

---

[5] Some FAMOS family members had no system processes at all.

code. Regardless of what environment the module is bound to, the representation of the module, once instantiated, is accessible to the translator. Therefore, should the initialization code of a later module invoke operations provided by an earlier module, the translator can carry out the specified actions on the host representation of the earlier module. Initialization clauses could not contain forward references, as this would imply the use of objects not yet instantiated.

Once a module has been instantiated, its host components are available to participate in system generation. For example, an address space manager instantiated at one level could be invoked to build address translation tables for subsequent levels. In general, the *system generation* activities of one level are triggered by the *initialization* activities of higher levels.

One of the system instantiation phases I have lumped under system generation is *downloading*. After all of the target code and data objects have been created, they must be transmitted to the target machine via some storage or communication medium. The order of transmission must be explicitly programmed. This can be achieved by invoking the downloading code from the initialization clause of the outermost module of the system. Each level would provide a downloading procedure for its level, which would clean up its own data structures, call the downloading procedure of the level below, then download the programs it has been given to manage.

We can now program the initialization, generation, and startup of the first level of a PDP-11. We will need a specification of the target hardware environment in which the bootstrap loader will run, and host facilities for placing the bootstrap loader onto the tape. To be complete, I also insert the module specification for the compiler, although it doesn't contain anything interesting:

```
Module<host> compiler is
   requires linker
   provides type procedure, variable, task, etc.
   . . .
end module compiler
```

```
Module PDP-11
requires Linker
   provides Mp, CommArea, BootEnvironment

   var Mp: CoreBlock[0:177777]     -- full configuration

         module CommArea is
            requires Linker
            provides TrapSeg, IOSeg, TrapVectors, Devices, etc.

            var TrapSeg: segment (at 0 in Mp, length 240, volatile )
             -- trap vector declarations go here
            var IOSeg: segment
                 ( at 160000 in Mp, length 20000, volatile )
             -- device register declarations go here
            . . .
         end module CommArea

         type Boot Environment is        -- an environment
            requires Linker
            provides ProgramRegion, LogicalSegmentSet, Startup

            var ProgramRegion: segment
                           (at 1000 in Mp, length 512 bytes)
            var LogicalSegmentSet: LogicalMemory
                           := (TrapSeg, IOSeg, ProgramRegion)
            condition<Boot Environment> Startup
         end type Boot Environment
end module PDP-11
```

TrapSeg and IOSeg are descriptions of the physical machine. They are listed as volatile so that the linker will not produce a host representation for their contents, nor allow initial data to be stored in them during translation.

I have written BootEnvironment as a type rather than a module, because a single operating system could in fact have several boot blocks residing on different secondary storage devices. The condition *startup* defined in BootEnvironment represents the mechanism by which the primitive loader residing in ROM on the bare PDP-11 transfers control to the program residing in the BootEnvironment.

Observe that elaborating the above module definition does not require any interaction with the target machine. The only concrete actions that would be taken are the creation of host machine descriptors for two volatile segments, and creation of a CoreBlock to track the relocation of segments.

Now that we have a type definition for a bootstrap environment, we can create one and provide the host machine facilities necessary to support it.

```
Module Bootstrap
   requires PDP-11, DecTapeImageFile
   provides BootEnv, DownLoad, HostDT

   var<host> HostDT:  DecTapeImageFile

   var BootEnv: BootEnvironment
   proc<host> DownLoad =
         init (HostDT)
         Link (BootEnv.ProgRegion)
         Write (HostDT, BootEnv.ProgRegion, Block = 0)
end module Bootstrap
```

Elaborating this module would result in creating a segment to hold a boot block, a logical memory descriptor, a file to hold the generated system for transfer to a dectape, and a host procedure for downloading BootEnv. It would also place an instance of the condition *Startup* in the host representation of BootEnv. Again, nothing has actually been done on the target machine.

We have now completely described a very simple virtual machine, consisting of a 32K word address space that contains only 256 words of usable program region. If we were to concatenate the definitions of the loader, compiler, PDP-11, and BootStrap, and elaborate them, we would be creating a virtual machine. We could then proceed to create a tiny program and place it in BootEnv, and call DownLoad to place it on tape. The tiny program would consist entirely of a handler for the condition BootEnv.Startup.

We shall now use BootStrap to support a loader that can fetch an arbitrarily large program from the DecTape. We presume that one of the device control registers declared in PDP-11 is named TargetDT, and controls the DecTape.

```
Module DecTapeLoader is
   acquires BootStrap.BootEnv, Bootstrap.Download
   requires PDP-11, HostDT
   provides SimpleEnv, DownLoad, Startup

      module SimpleEnv is          -- an environment
         provides ProgramRegion, LogicalSegmentSet

         var ProgramRegion: segment
                        (after BootEnv.ProgRegion in Mp)
         var LogicalSegmentSet: logical memory
                  := (TrapSeg,IOSeg,SimpleEnv.ProgramRegion)

   Condition<SimpleEnv> Startup

   var<BootEnv> RegionLength, RegionBase

   proc<host> DownLoad =
         Link (SimpleEnv.ProgRegion)
         RegionLength := length (SimpleEnv.ProgRegion)
         RegionBase := Base (SimpleEnv.ProgRegion)
         BootStrap.DownLoad()
         Write (HostDT, SimpleEnv.ProgRegion, Block=1)
         end

   handler<BootEnv> for BootEnv.Startup =
         TargetDT.Read(block = 1, length=RegionLength,
                        Address = RegionBase)
         raise DecTapeLoader.Startup
         end

   end module DecTapeLoader
```

SimpleEnv provides a program region of unspecified size, located immediately following the boot block in primary memory. A program residing there will be allowed to use the trap segment and the I/O segment, but not the boot block. One of the things a program residing there can do is handle the condition "DecTapeLoader.Startup".

The DecTapeLoader records the size and starting address of the SimpleEnv program region in a pair of target variables, RegionBase and RegionLength. Rather than writing their values out onto the tape separately, these values are recorded right in the BootEnv. Observe how this happens: the host procedure DecTape.Download links the program region and records its attributes in the two variables *before* calling BootStrap.Download. After the bootblock is recorded on the tape, DecTapeLoader.Download puts the contents of SimpleEnv.ProgRegion on tape starting at block 1. DecTapeLoader enables a handler for BootEnv.Startup, which uses RegionBase and RegionLength to move SimpleEnv.ProgRegion from tape to primary memory.

Elaborating the module DecTapeLoader still does not require any interaction with the target machine. Each of the target-machine objects created is represented in

some segment, which in turn is represented as a host machine file. Even enabling the handler for BootEnv.Startup only requires proper placement of the compiled code in the tiny program region. The module definition does not call for any target machine procedures to be invoked.

(This example does not attempt to solve the problems of allocating tape space in a modular fashion and reusing target machine memory for successive layers. Such issues will be addressed in Chapter 5.)

So far we have declared two host machine procedures: BootStrap.DownLoad and DecTapeLoader.DownLoad. Each procedure embodies the system generation program for the corresponding machine level, namely linking the managed segment and moving both the customized machine and the managed segment to tape. However, we have not called for either of these procedures to actually be invoked. Below we will do so, after first elaborating an arbitrary user program:

```
   .
   .
   .
        Module<SimpleEnv> UserProgram is
           requires SimpleEnv.Startup
           . . .
           handler for SimpleEnv.Startup is
           . . . -- This is user's main program on PDP-11
        end module UserProgram
    -- Outermost initialization generates a system :
   Begin
        DectapeLoader.DownLoad()
   End
end module System
```

The user's program is a single module, bound to SimpleEnv. Elaborating that module places its code and data in SimpleEnv.ProgramRegion. When the translator elaborates the outermost initialization clause, it invokes the DecTapeLoader's download procedure, which precipitates the entire system generation process. To get an overview of that process, we shall first abstract the modules defined in this section and combine them into a single, abbreviated system description:

```
module system is
    module<host> Linker is
        provides Segment, Section, CoreBlock,
                 LogicalNameTable, LogicalMemory
            type<host> segment is
                requires Section
                provides Insert, Link, Copy

    Module<host> compiler is
        requires linker
        provides type procedure, variable, task, etc.

    Module PDP-11
        requires Linker
        provides Mp, CommArea, BootEnvironment
            module CommArea is
                requires Linker
                provides TrapSeg, IOSeg, TrapVectors,
                         Devices, etc.
            type Boot Environment is -- an environment
                requires Linker
                provides ProgramRegion, LogicalSegmentSet,
                         Startup

    Module Bootstrap
        requires PDP-11, DecTapeImageFile
        provides BootEnv, DownLoad, HostDT

    Module DecTapeLoader is
        acquires BootStrap.BootEnv, Bootstrap.Download
        requires PDP-11, HostDT
        provides SimpleEnv, DownLoad, Startup
            module SimpleEnv is  -- an environment
                requires Linker
                provides ProgramRegion, LogicalSegmentSet

    Module<SimpleEnv> UserProgram is
        requires SimpleEnv.Startup
```

We can trace the flow of control through this system description by expanding the various downloaders and condition handlers in line, as follows:

- Link (SimpleEnv.ProgRegion)
- RegionLength := length (SimpleEnv.ProgRegion)
- RegionBase := Base (SimpleEnv.ProgRegion)
- init (HostDT)
- Link (BootEnv.ProgRegion)
- Write (HostDT, BootEnv.ProgRegion, Block = 0)
- Write (HostDT, SimpleEnv.ProgRegion, Block=1)
- Person copies HostDT to real DecTape
- Person carries tape to a PDP-11
- *Person sets boostrap address, presses start switch*
- Bootstrap ROM loads DecTape block 0 into location 1000 ff.
- TargetDT.Read(block = 1, length=RegionLength, Address = RegionBase)
- raise DecTapeLoader.Startup

Thus we see that we have successfully written a program to create and run an operating system, albeit a trivial one. The fact that we can trace the entire system generation process gives us confidence that the system description is comprehensive, as we had hoped.

3.7. Summary

We have made the concept "execution environment" a concrete entity in system descriptions, by using it to define the interface between the implementation language and the system. We have defined a notation for explicitly programming the associations between source program units and execution environments, so as to support both multi-environment modules and multi-module environments. We found the notation sufficient to express a variety of type management styles in multi-environment systems, including monitored types. By introducing the host machine as one of an operating system's execution environments, we were able to program initialization and system generation activities as part of the modules that manage the generated programs. The resulting system description is a complete program to create the operating system.

The **acquires** clause introduced in section 3.2 facilitates programming incremental machine designs that span multiple environments. By supporting partial concealment, it facilitates static representation and checking of the *uses* relation among system components. In particular, it provides a way for a module to obtain *exclusive access* to an *environment* defined below it, so that the module controls what code can be placed in that environment.

The toy problem solved in the last section was contrived to illustrate system generation, and is not by itself a demonstration of the value of the methodology. In the next three chapters we shall assess the power of the concepts and notations I have introduced, by applying them to the system description problems identified in section 2.2.

CHAPTER 4
RELATING TO HARDWARE

In this chapter we apply the proposed methodology to the description of hardware device communication facilities. Interrupt handling and device handling programs are some of the most likely programs to be written in assembly language rather than a higher-level language. Interrupts themselves don't fit into conventional synchronization methodologies. Priority levels likewise are foreign to high-level languages. Because short routines involving volatile storage are less likely to benefit from optimizing compilers, the explicit control available in assembly language seems to out weigh the benefits of strong typing and notational uniformity.

Device registers also pose a name control problem quite apart from interrupts and priorities. If the device control registers are considered to be a set of objects provided by the module "hardware", then each module that uses and conceals a device register must build on top of the hardware module, rather than simply importing the device control register, in order to ensure that no other module has access to that device. This imposes a total ordering on the modules that handle interrupts, an ordering which must be made to fit other ordering constraints imposed by interrupts and priorities.

Modula and Concurrent Pascal both abstract away from the interrupt level altogether, providing the programmer instead with a high-level synchronization construct, the *monitor* (or *interface module*, in Modula), with which to describe device communication. In section 2.2.1 I argued that each of these languages substantially encumbered system design by burying interrupt handling in the language runtime system.

A *fit* language for describing device interactions must contain both a suitable construct for defining interrupt routines, and a suitable means for describing the effect of priority levels. The language must also be highly *transparent*, in order not to impose overhead or constrain design.

This chapter presents the design of a device communication subsystem for a DEC VAX-11, similar to the system used in the VAX/VMS operating system. The individual device control registers and interrupt vectors are declared in a module that is responsible for the entire machine description. This module provides an interrupt module for each device, containing both its interrupt vector and its control and data

registers. The interrupt vector is represented in the system implementation language as an exceptional condition, as defined by Levin [Levin 77]. The language defines the minimum set of features an interrupt module must define. Each interrupt module is acquired by the module that is to manage the device. That module enables one handler, permanently, for responding to the interrupt. The handler is written in the form of an ordinary procedure.

Interrupt routines usually communicate with other system components through shared variables. Synchronization can be implemented either by manipulating the interrupt priority level of the processor, or by masking interrupts from individual devices. A language for describing such synchronization should allow the system designer as much control over priority and selective masking as reasonable, but need not support unreasonable control. I postulate that in a reasonable program the priority at which each statement executes should be determinable at compile time. I propose a locking protocol that is only a slight restriction from primitive data locks, and show how to compute the priority for each program statement that is both necessary and sufficient to implement the requested locks.

I represent selective masking by defining the action **defer** on a condition, defined in terms of Levin's primitive actions on conditions. I again show how to determine the necessary and sufficient priority for each statement, this time including knowledge about which handlers might be deferred.

The priority determination method could be automated and incorporated into a software development control facility. For the purposes of this thesis I indicate the priority of each statement explicitly, so that the reader can see if a change to one part of the system might change the necessary and sufficient priority of some other part of the system. Whether the priorities are computed by the language system or simply verified sufficient, incorporating priority manipulation into the language provides significant assistance in producing reliable software, without unduly constraining the system design space.

4.1. Interrupts as Exceptional Conditions

I briefly introduced Levin's *exceptional condition* mechanism in section 3.6, using it to describe bootstrap loading. The mechanism he defines is very general, suitable for programming unusual function returns, reporting data structure inconsistencies, handling arithmetic overflow, and soliciting the return of unneeded resources when the pool runs low, as well as interrupt handling, all in a multiprogramming context. He is careful to point out that specific applications do not need the full generality of the mechanism, and would be more efficient if only the needed features were implemented. Also, he conjectures that certain usage patterns will occur so frequently as to justify special language features that implement them even more efficiently. He envisions a large system containing several different exceptional

condition reporting facilities, unified by a common syntax and semantics, but each supporting only a subset of the mechanism.

I shall define a subset of Levin's handler mechanism that models the behavior of interrupts and interrupt routines, and gracefully supports customary programming methods. I shall define a special language feature, **defer**, to model the effect of turning off interrupts from a particular device, in terms of the basic operations on conditions.

An interrupt is an event signalling the occurrence of some condition on some object or process. For hardware devices, it normally signals the completion of some command, or the occurrence of some asynchronous event, such as the expiration of a time interval or the arrival of a message from a network. A memory violation interrupt signals a condition on the currently executing process. However, for the purposes of this chapter we will limit our discussion to conditions on objects. A **handler** for a condition is a procedure designed to respond to the occurrence of the event. When a device or program *raises* a condition (signals its occurrence), the handler for the condition is executed. The handler may use variables defined in the scope in which it is declared, and may also receive parameters from the device or program that raises a condition. (Device status registers would be accessed as shared variables, not as parameters. Certain hardware conditions in VAX, such as memory violations, push parameters on the stack.) There may be more than one handler for a condition, but a handler is only invoked *if it is* **enabled** for that condition. A handler may be enabled for the duration of some program statement, such as a loop or procedure call, or may be enabled for the lifetime of the object on which the condition is defined. An interrupt condition may have at most one handler enabled for it at one time. Normally that handler will be enabled permanently.

Levin's mechanism dictates that when a condition occurs, its handler is invoked immediately. Since the CPU interrupt priority level may delay the invocation of a handler, we must make some kind of accommodation. I choose to define the interrupt dispatcher of the CPU as an intermediary between the hardware devices and the software handlers. Each device may raise a condition, which the dispatcher handles by setting a flag indicating that it occurred. Whenever the interrupt priority level falls, the dispatcher checks its flags to see what conditions should be signalled. The condition thus signalled is different from the one the device raised, because it reflects both a property of the device and a property of the interrupt priority level register, e.g. "the clock has ticked recently and priority is now below level 15". In practice the priority is used only for controlling the urgency of computation and for mutual exclusion. I will add notations for explaining these properties later. Meanwhile, we shall assume that by coincidence no interrupt ever occurs during the execution of another interrupt handler.

Let us define a language system interface for interrupt vectors and devices. The language will provide the following constructs to users of interrupt conditions:

handler for *condition* = *procedure body*
raise *condition*
defer *condition*

The **handler** clause declares and enables a handler for the condition. For interrupt conditions it shall only appear in a permanent module instance, making it a permanently enabled handler. The language system must know what parameters, if any, will be passed by the signaller to the handler, and must give the exception handling mechanism the address of the handler. For software interrupts, the user of the condition may also **raise** it. The language system must invoke an operation on the exception mechanism when this occurs, transmitting the necessary parameters. Many device registers support operations to suspend interrupts from an individual device, regardless of the current priority level. To **defer** a condition means to postpone invoking the handler for the duration of some sequence of statements. To support this, the language system will need operations to mask and unmask the interrupts from that device at the beginning and end of the statement sequence.

An **interrupt module** shall have the following form:

```
interrupt module module name is
    provides condition, priority[, defer, raise]

    condition ( parameter list )
    const priority = integer
    proc set ( identifier : address ) = procedure body
                -- called to enable a handler

        -- The following are optional. Their presence indicates
        -- that the corresponding language features are supported.
    proc mask = procedure body
    proc unmask = procedure body
    proc signal ( parameter list ) = procedure body
end module module name
```

The facilities that are to be used only by the language system, namely *set*, *mask*, *unmask*, and *signal*, are not provided, and so cannot be invoked directly by other modules. Instead, the module provides the corresponding *language feature*, leaving it to the language system to perform the translation. As with other language support modules, an interrupt module may provide any other facilities that are appropriate, such as access functions for the device registers associated with the interrupts.

4.2. VAX Interrupt and Device Hardware

The VAX architecture provides 16 levels of hardware interrupts, and 15 (lower) levels of programmable interrupts. Interrupt handlers may execute either on the kernel stack (provided that no other handler is running at the time the handler is dispatched) or on the interrupt stack. Each interrupt routine begins execution at the same priority level as the interrupt itself, but may raise or lower that priority. However, the priority at the end of the handler execution must be at least as high as when it started, because the REI (return from exception or interrupt) operation will fail if completion would cause an increase in priority. Also, there is a kernel stack for each process in the system, and the context swapping routines must use the REI instruction, so any routine that calls the context swapping code must be executing at a lower priority level than the level of the context swapping code.

Device control registers on VAX are accessible through ordinary memory read and write operations, just like on PDP-11's. Each device is assigned a separate interrupt vector, so that interrupt routines don't need to poll several devices to find the origin of an interrupt. The interrupt vectors are located in the "system control block", which is identified to the CPU by the "system control block base" register.

4.3. VAX Device Communication Subsystem

We can now program the device registers and interrupt vectors for VAX. For convenience we shall draw upon the simple system described in section 3.6 so as not to become mired in repetitive detail. We shall build an execution environment that supports all devices and can access all of primary memory, through a page table that provides an identity map for primary memory pages, and a suitable address range for the device registers. One practical use for such an environment would be an interactive loader that could select system components from directory-structured devices, in response to console commands. Here is the overall structure of the system:

```
module system is
    module<host> Linker is
    provides Segment, Section, CoreBlock, LogicalNameTable,
             LogicalMemory

    Module compiler is
       requires linker
       provides type procedure, variable, task, etc.

    Module VAX
    requires Linker
       provides Mp, BootEnvironment, Devices
          type Boot Environment is -- an environment
             requires Linker
             provides ProgramRegion, LogicalSegmentSet,
                      Startup
          type Devices is
             provides Console, Disk1 .. DiskN,
                      DiskController, etc.

    Module Loader is
       requires VAX, HostDisk
       provides SimpleEnv, DownLoad, Startup
          Module Bootstrap
             requires VAX, PseudoDisk
             provides BootEnv, DownLoad, HostDisk

    Module<SimpleEnv> IO is
       provides TransferRequest, IOPostQueue, . . .
```

The components that pertain to device communication are:

- VAX: declares the device registers
- Devices: defines the software control block containing all interrupt vectors, and pairs up each interrupt vector with the corresponding device registers in an interrupt module.
- IO: declares the queues through which input and output request blocks pass going to and from the devices, and declares procedures and condition handlers that process device commands.

The linker and compiler perform functions analogous to their counterparts in the PDP-11 loader. This loader is capable of placing a core image in primary memory and raising the condition *startup*, to be handled by the program residing in that image. (This condition is implemented by setting the initial program counter to the address of the handler, rather than through the interrupt mechanism.) Most of the details of the loader are irrelevant to this example, and will be omitted.

4.3.1. The Module VAX

The module VAX defines the physical characteristics of the hardware, including the CPU registers, the primary memory and device communication memory, and the interrupt vectors. This is its general form:

```
module VAX is
    requires linker
    provides Mp, BootEnvironment, Devices

    var Mp, IOMem, IOSeg, DeviceRegisters- . . .

    module CPU is
        provides SCBB, . . .

    type BootEnvironment is
        requires PTType, Linker,
        provides ProgramRegion, LogicalSegmentSet, PageTable, Startup


    type Devices(SCBSegment: Segment) is
        provides ConsoleRec, ConsoleSend, DiskController,
                    MagTapeController, SoftInt1 .. SoftInt15, . . .

        interrupt module ConsoleRec

        interrupt type DiskControllerType is
        . . .
        interrupt type SoftIntType(i:SoftIntIndex) is
        . . .
    end type Devices
end module VAX
```

IOMem is the range of physical addresses set aside for device control and status registers. I declare the segment IOSeg to completely fill that range of addresses. All of the device control registers are bound to IOSeg, as follows:

```
var Mp: CoreBlock(1 Megabyte)
var IOMem: CoreBlock (Base 512M, length 1M)
var IOSeg: segment (at 0, length 1M, in IOmem, volatile)
var<IOSeg> DeviceRegisters : record
    var ConsoleRegisters
    var DiskRegisters
    var TerminalRegisters
        etc
```

Bootstrap loading on VAX is performed through a console that includes an LSI-11 microcomputer and one or two floppy disk drives. The console can load an entire binary file from a floppy disk into primary memory while the VAX CPU itself is halted. To simplify this system description, I assume a console command file that initializes the system page table pointer to zero, implying that the first part of every load file should be a page table. I document this design decision in the definition of BootEnvironment, which contains a segment for the page table and one for the remainder of primary memory. BootEnvironment also defines the condition *Startup*, which is implemented by a console command to start execution at the address of

the handler code. BootEnvironment is a type rather than a module so that a system may create more than one core image, for different purposes.

```
type BootEnvironment is
     requires PTType, Linker,
     provides ProgramRegion, LogicalSegmentSet, PageTable, Startup

     var MapSeg: segment (at 0, length 16K, in Mp)
     var ProgramRegion: segment (at 16K, in Mp)
     var<MapSeg> PageTable: PTtype :=
                         (MapSeg.BootEnv.ProgramRegion, IOSeg)
     var LogicalSegmentSet: LogicalMemory :=
                                   (MapSeg,BootEnv.ProgramRegion,IOSeg)
end type BootEnvironment
```

Devices is a type rather than a module because VAX allows the interrupt vectors to reside anywhere in primary memory. They are made known to the CPU through the System Control Block Base (SCBB) register. The record SCB (for System Control Block) defines names for each of the interrupt vectors:

```
var<SCBSegment> SCB:packed record
     aligned mod 512
     var unused1: interrupt vector
     var MachineCheck: interrupt vector (IntStack)
     var KernelStackNV: interrupt vector (IntStack)
     var PowerFail: interrupt vector (KernStack)
     . . .
     var SoftIntVec: array [SoftIntIndex] of interrupt vector (KernStack)
     . . .
     var ConsoleReceive: interrupt vector (IntStack)
     var ConsoleTransmit: interrupt vector (IntStack)
     . . .
     var DiskController: array [DiskIndex] of interrupt vector (IntStack)
end record SCB
```

The type *Devices* acquires *DeviceRegisters* so that the registers will be used only through the interrupt modules. Each interrupt module acquires the device control and status registers and the interrupt vector it needs. For example, here is the interrupt module for the Console Receiver:

```
interrupt module ConsoleRec
     acquires ConsoleReceive, ConsoleRecReg
     provides condition, priority, defer, Console receive operations

     condition (no parameters)
     const priority = 14HEX
     proc mask is
          ConsoleRecReg.mask := true
     proc unmask is
          ConsoleRecReg.mask := false
     --  other operations on Console Receive go here
end module ConsoleRec
```

Because there is an interrupt vector for each disk controller on VAX, there must be a separate interrupt module for each. Therefore, I define an interrupt type giving the form of the module, and then declare as many instances as I need:

```
interrupt type DiskControllerType is
    acquires SCB.DiskController,DeviceRegisters.DiskController
    provides condition, priority, defer, Disk command ops

    . . .

end module DiskControllerType

var DiskController: array [DiskIndex] of DiskControllerType
```

Software interrupt vectors support raising conditions as well as handling them:

```
interrupt type SoftIntType(I:SoftIntIndex) is
    acquires SoftIntVec
    provides condition, priority, raise

    condition (no parameters)
    const priority = I
    proc signal =
        CPU.SoftwareInterruptRequestRegister := I

    . . .

end type SoftIntType
var SoftInt1: SoftIntType(1)

    . . .

var SoftInt15: SoftIntType(15)
```

Although the text of the type *devices* is rather lengthy, the only primitive objects instantiated in it are the interrupt vectors themselves. The rest of the type simply defines the interrupt conditions. Creating an instance of *devices* creates an SCB in the environment of the instance. Here is an abbreviated definition of the entire module *Devices*, to show how the pieces fit together:

```
type Devices(SCBSegment: Segment) is
     provides ConsoleRec, ConsoleSend, DiskController,
                    MagTapeController, SoftInt1 .. SoftInt15, . . .
     var<SCBSegment> SCB:packed record
          aligned mod 512
          var unused1: interrupt vector
          var MachineCheck: interrupt vector (IntStack)
          var KernelStackNV: interrupt vector (IntStack)
          var SoftIntVec: array [SoftIntIndex]
                              of interrupt vector (KernStack)
          var ConsoleReceive: interrupt vector (IntStack)
          var ConsoleTransmit: interrupt vector (IntStack)
          var DiskController: array [DiskIndex]
                              of interrupt vector (IntStack)
          . . .
     end record SCB

     proc SetSCB is
          CPU.SCBB := address of SCB

     interrupt module ConsoleRec
          acquires ConsoleReceive, ConsoleRecReg
          provides condition, priority, defer, Console receive operations

          condition (no parameters)
          const priority = 14HEX
          proc mask is
               ConsoleRecReg.mask := true
          proc unmask is
               ConsoleRecReg.mask := false
     end module ConsoleRec

     interrupt type DiskControllerType is
          acquires SCB.DiskController,DeviceRegisters.DiskController
          provides condition, priority, defer, Disk command ops

          . . .
     end module DiskControllerType

     var DiskController: array [DiskIndex] of DiskControllerType
     . . .
     interrupt type SoftIntType(I:SoftIntIndex) is
          acquires SoftIntVec
          provides condition, priority, raise

          condition (no parameters)
          const priority = I
          proc signal =
               CPU.SoftwareInterruptRequestRegister := I
          . . .
     end type SoftIntType
     var SoftInt1: SoftIntType(1)
     . . .
     var SoftInt15: SoftIntType(15)
     . . .
end type Devices
```

4.3.2. The Module Loader

The loader must define a useful execution environment and arrange to have it transported from the host to the target machine. It creates a **BootEnvironment**, and places an instance of **Devices** in it. *PageTable* contains an identity map of primary memory followed by a map of IOSeg. This defines the virtual addresses of the device registers, and makes it easy to compute the physical address of SCB for recording in SCBB.

```
module Loader is
    requires VAX, HostDisk
    provides SimpleEnv, Devices, DownLoad, Startup

    var SimpleEnv: Boot Environment
    var<SimpleEnv> Devices: VAX.Devices(SimpleEnv.ProgramRegion)
    condition Startup
    handler<SimpleEnv> for SimpleEnv.Startup =
        SetSCB
        raise Startup
```

4.4. Synchronizing Interrupt Routines

Interrupt modules supporting **conditions** provide an adequate mechanism for implementing interrupt routines for a strongly typed language. However, interrupt routines ordinarily communicate with other portions of a system through shared variables. The procedures that access them typically protect them from *race conditions* by raising the interrupt priority levels high enough to block competition from interrupt handlers, or by suspending interrupts from individual devices. A *fit* language for synchronizing interrupt routines would provide a great deal of control over the interrupt priority level, while at the same time providing as much help as possible in verifying mutual exclusion for shared variables.

Simply providing direct access to the interrupt priority level (IPL) register is unacceptable, because the language system cannot give any assistance in verifying synchronization constraints. Instead, the program ought to state which variables are to be locked, and when. I shall define a set of primitives for locking individual variables, and show how to implement the locking primitives as operations on the IPL register, such that the interrupt priority at any give time is both necessary and sufficient to implement the locking commands.

It is reasonable to expect that for any given statement of a program, the set of variables that are to be locked during its execution should be determinable at compile time. Interprocedural data flow analysis techniques allow us to determine which variables *might* be locked during a given statement; prudence dictates that they *must* be locked for that statement. This restriction allows the priority requirements to be determined at compile time.

Masking and unmasking individual interrupts can also benefit from language support. Data flow analysis can determine which handlers *might not* be masked during a given statment; by assuming they *will not* be masked, we can again determine the necessary and sufficient priority for mutual exclusion.

Data locks and handler masking are unacceptably clumsy programming tools. I shall introduce the language construct **defer**, defined in terms of primitive operations on **conditions**, and implemented with *mask* and *unmask*. Similarly, many reasonable synchronization constructs can be implemented as data locks. I choose a form of **critical regions**, which names the variables to be locked, and which can span either procedures or entire modules. I shall demonstrate that the construct is immune to the (non)problem of nested monitor calls [Parnas 78b].

Using critical regions and interrupt condition handlers, I shall show how to program a device communication subsystem for VAX, much like the one actually present in VAX/VMS. By imbedding the priority and masking functions in the implementation language, I can express the needed synchronization in abstract terms, while retaining reasonable control over changes in priority.

4.4.1. Analyzing Programs That Use Locking Primitives

One can describe the mutual exclusion requirements of a system of programs by inserting explicit *lock* and *unlock* operations on individual variables. In general this technique is unacceptable, because two processes that lock the same set of variables in different orders may become deadlocked. This problem can be avoided by imposing a strict ordering on variables, such that a process must lock the variables it uses in a predetermined order. Because we intend to implement locks with priorities, we know that the ordering implied by priority levels will impose a locking order on the variables. Therefore, the excess generality of locking primitives in this situation is harmless. In fact, the problem of implementing locking with priorities reduces to the problem of assigning variables to priority levels. To lock a variable, one raises the current priority level to the level of that variable.

Because interrupt routines must share a single call stack, an interrupt routine must be allowed to run to completion once it has begun execution. To block an interrupt routine, one must prevent it from being dispatched at all. So, to lock a variable X, one must determine all the interrupt routines that might access X, and choose a priority level sufficiently high to prevent any of those interrupt routines from being dispatched.

I assume that a variable will be locked if it is to be used. I do not prevent the programmer from using a variable without locking it, nor do I detect such uses. One can determine all of the variables a handler locks by forming the transitive closure of the *calls* relation, and for each procedure in *calls**(H), mark the variables

locked in that procedure as locked by handler H. The priority level needed to lock a variable V is the maximum initial priority among all handlers that lock V.

To implement a completely dynamic locking mechanism, one would add to the state description of a process a *locking set* naming the variables currently locked. The procedures *lock* and *unlock* would change the set, and adjust the interrupt priority level to fit. However, this mechanism would be entirely too expensive for use in the interrupt routines of an operating system. Furthermore, a program in which the priority of a statement could not be determined until run time, would be hard to understand. Therefore, I make the following restriction:

> If a variable V *might be* locked when control reaches statement S,
> then variable V *will be* locked when control reaches S.

Later, when I propose a specific language construct for synchronization, I will be making more restrictions. This one simply limits the class of constructs I am interested in supporting.

We determine what variables might be locked at a given point P by interprocedural data-flow analysis. A variable V might be locked at P if there exists a path to P, from the beginning of any handler or process, in which the variable is locked and not subsequently unlocked. This is nearly identical to determining the set of *available expressions* at a point P, if one views *lock*(V) as defining V and *unlock*(V) as killing V. The only difference is that for available expressions we require *every* path to define the expression.

However, we now discover that the restriction above might be too strong. A utility procedure that makes no reference to any shared variable might be called from two places: one which locks some variable V, and one which does not. The restriction would require V to be locked for both calls. Instead, if we view each invocation of a procedure as a separate copy of the procedure, the conceptual problem disappears. When we imbed the lock operations in a more structured synchronization tool, we will see that actually making separate copies is unnecessary.

Having determined the set of variables that might be locked at each point P, a compiler can generate instructions to adjust the priority levels at each *lock* and *unlock* operation, such that the priority is always high enough to protect the locked variables, but never higher than necessary.

4.4.2. Masking Individual Interrupts

Most device control registers contain a flag that determines whether or not the device will interrupt the CPU when it completes a command. An error recovery procedure could obtain mutual exclusion for the device register by turning off interrupts from that device. Although priority manipulation is a more common form of synchronization, I wish to demonstrate that this form, too, can benefit from language support.

Consider a program containing both locking primitives and masking primitives (*mask* and *unmask*). For a dynamic implementation one would add a *masking set* to each process state descriptor. The necessary and sufficient priority at any time would be the maximum initial priority of any *non-masked* handler that, if dispatched, might lock one of the variables in the *locking set*. Each locking and masking operation would recompute the priority to accommodate the changed set.

Again, we would like to make the priority determination at compile time. By interprocedural data flow analysis we can determine the handlers that will have been masked when execution reaches a given point P. We again view different invocations of a procedure as separate copies, since one invocation might mask a given handler, and another not. For each invocation, if there is uncertainty, we assume that the handler in question will *not* be masked. Knowing the handlers that will be masked at each point, a compiler can generate the necessary IPL register settings to protect the locked variables.

4.4.3. A Synchronization Notation

Individual data locks are sufficiently primitive that many well-known synchronization mechanisms can be built out of them. For the purposes of this thesis I choose a very simple one, just elaborate enough to support the programming example I will be presenting. I see great opportunity for further research to discover which synchronization constructs can be simply implemented with priorities.

I choose a **critical region** construct that names the variables the region is to protect:

```
crit ( A, B, C)
    statement-sequence
end crit
```

I also allow an entire module or type to be declared a critical region:

```
module M is
    requires . . .
    provides . . .

    crit ( A, B, C)

    . . .

end module
```

All critical regions that protect a given variable must mutually exclude one another. A critical module or type is simply an abbreviation for making every procedure body defined therein critical. Critical regions have dynamically nesting scope. Consider procedures P and Q, where P calls Q. Any variables protected by a region spanning the call of Q shall be protected for the duration of the call as well:

```
var A, B
proc P is
    crit ( A )
    ...
    call Q
    ...
    end crit

proc Q
    crit ( B )
    ...
    end crit
```

The variable A is protected throughout each invocation of *P*, even while *P* is waiting for the completion of Q.

To represent masking of interrupts from particular devices, I choose to define a construct to **defer a condition**. Levin's general condition mechanism allows a handler to be enabled for the duration of any statement, written as follows:

<*statement*>[*condition-name* : *handler-body*]

A handler enabled this way supercedes handlers enabled farther down the call stack. Also, if there is a handler statically enabled for the same condition, this handler takes priority over the other handler. I now define the operation *defer* by the following rewrite rule:

<*statement*>[**defer** *condition-name*]

becomes

```
begin
    boolean flag := false;
    <statement> [condition-name : flag := true];
    if flag then raise condition-name
end
```

This rewrite rule describes what a typical device actually does when it is masked: upon completing a command it sets an internal flag; if that flag is set when the device is unmasked, it sends an interrupt.

4.4.4. Implementing Critical Regions

Critical regions could be implemented dynamically by maintaining a multi-set for each process listing the variables currently protected by that process, and locking or unlocking individual variables as appropriate. However, critical regions can easily be translated directly into priority manipulations.

The function *calls**(H) tells which variables a handler locks, by revealing which critical regions it enters. The priority class of a variable follows from this information as before. The priority of a given critical region is determined from the priority classes of its variables *and* the priority of the dynamically enclosing region. The entry code for a critical region shall push the current IPL value on the execution stack, and replace it with a higher priority, if needed, to protect the variables of the region. Exit code simply restores the old value of the IPL register.

This implementation of critical regions removes the need to make separate copies of a procedure that is invoked from different locking contexts. Each caller of a given procedure will set the IPL register sufficiently high to protect the variables locked prior to the call; the called procedure will maintain a priority at least that high.

A system designer may need to know at what priority each critical region actually runs, in order to assess response time for time-critical handlers. A software development control facility could easily display the determined priorities in the source text for this purpose. Lacking that, a language designer could ask the programmer to specify the priority of each critical region, and have the translator verify the mutual exclusion. For the remainder of this chapter I will specify the priority of a region in the source text whenever it is significant.

4.4.5. Implementing Defer

An *interrupt module* will define the operations *mask* and *unmask* if the device being described is to support the *defer* construct. To allow for nested **defer** regions, the translator would supply a counter for each process and each handler, to record the number of dynamically nested **defer** regions entered for a given handler by the given process.

Defer regions also affect the priority computations, however, and here we discover a minor unpleasantness. Although we can determine, for each critical region, exactly which handlers are certain to be masked when executing in the region, we must either generate separate copies of procedures invoked from different masking contexts, or assume the handlers in question are not locked.

4.4.6. Coordination with Process Multiplexing

The design described so far will fail for VAX if the operating system's process multiplexor suspends the currently executing process and begin executing another during any critical region, because the interrupt priority level is maintained on a per-process basis. VAX/VMS deals with this problem by doing context swapping at a priority below all device-handling priorities, so that all critical regions that involve devices will block context swapping as well. To incorporate this idea into the current design, we specify that every critical region will execute at a priority at least as high as the context switching priority, so that a critical region can synchronize ordinary processes even if no interrupt handlers are involved.

4.4.7. Nested Monitor Calls

Monitors were involved recently in a controversy surrounding the "problem of nested monitor calls". The debate surrounds the question of whether a process calling monitor Y from within monitor X should be thought of as "leaving" monitor X. If not, then deadlock can result should another process call X from within Y. Parnas argues that the answer to the question depends entirely on the individual program. [Parnas 78b]. Modula avoids this alleged problem by forbidding all calls out of an interface module. This means that all communication between handlers must pass through ordinary processes, which is unacceptable.

A critical module, as I have defined it, strongly resembles a monitor. Let us consider how it would answer the question for the following example:

```
module M is
     acquires ConsoleReceiver
     requires Q
     provides P

     var A
     crit ( A, priority = ConsoleReceiver.priority )
     handler for ConsoleReceiver.condition is ...

     proc P =
          ...
          call Q
          ...
     end proc P

     ...
end module M


module N is
     provides Q

     var B
     crit ( B, priority = ContextSwap.priority )
     proc Q = ...
end module N


proc R =
     call P
     call Q
end proc P
```

Assume that the interrupt priority of the ConsoleReceiver is 14, and that Q is not called by any handler, directly or indirectly. (This makes the priority for N be the context switching priority, say 2). All of the code within module M would execute at priority 14 or higher. When procedure P calls procedure Q, priority would remain at 14. However, because the handler for ConsoleReceiver.condition does not call Q, the priority class for B is not affected by it. Thus, when R calls Q, the priority at which Q is executed can be lower than 14.

Suppose that the priority class of N were 10, rather than 2, because a new handler at priority 10 uses Q. That handler could not enter N during the execution of P because the IPL would be too high. Furthermore, no handler could enter P during the execution of Q, because that handler's initial priority would be less than or equal to 10, and therefore it would not be dispatched.

In general, a process executing in one critical region may enter another critical region without leaving the first, and not risk deadlock. The priorities used to implement the regions define a locking order for them, so that there cannot be a deadly embrace.

4.5. Device Drivers for VAX

VMS allocates four hardware priority levels and four software priority levels to I/O handling, in addition to one level of each for the interval timer. The general design paradigm for interrupt handling is that a device driver running at a software interrupt priority level maintains the request queue for a device, passing one command at a time to the hardware priority level. The hardware interrupt handler simply copies the status information into the command description block, and puts that block in the driver's queue of completed commands. At a still lower software interrupt level, VMS maintains a queue of all completed user requests, sending the completion notices on to the appropriate processes.

To illustrate device communication, I shall describe a subsystem that provides a command to initiate I/O transfers, and a queue of completed requests. A request is described in a Transfer Request Block (TRB). The overall structure of the subsystem is as follows:

```
module IO is
    acquires loader.devices
    provides IOPostQueue,TransferRequest, TRB

        module IOPostQueue is
            acquires Softint4
            provides IODone, SendIOPost, GetIOPost

        module IODoneQ11 is
            acquires Softint11
            requires DiskComplete, MagTapeComplete, . . .
            provides IODone11

    module IODoneQ10 is . . .
    module IODoneQ9 is . . .
    module IODoneQ8 is . . .

        Module DiskManager is
            requires IODoneQ11, SendIOPost
            provides DiskSubmit, DiskComplete

                type Driver(I: DiskIndex) is
                    acquires Devices.DiskController
                    requires IODone11
                    provides Submit, Complete

            var Cont: array [DiskIndex] of ref driver

    module MagTapeManager is . . .
    module NetworkManager is . . .
    module TerminalManager is . . .
```

The IOPostQueue collects the completed requests, raising a condition each time a new entry is placed in the queue. The type TRB has all of the data for a request, plus link fields for placing it in queues. The procedure TransferRequest examines the request and routes it to the proper device driver. The IODoneQ's administer four software priority levels for processing I/O completions. The individual hardware interrupt handlers each place the completed blocks in these queues, to minimize execution time spent at the hardware priority levels, thus reducing the chance of a missed interrupt. The disk manager contains an array of controllers, with a separate request queue for each.

4.5.1. Transfer Request Queues

The VAX/VMS hardware supports four indivisible operations on doubly linked lists: insert and remove, at head or tail. They might be defined in a standard prelude as follows:

```
type DequeDefinition[ T: type ]
    provides deque, deqelt, value
             InsertHead, InsertTail, RemoveHead, RemoveTail
             InResults, RemResults

    type Etag: [member, header]
    type EUnion ( tag: Etag ) = record
        var head, tail: ref EUnion ( any )
        case tag of
                member:[var value: T]
                header:[]
        end record

    type deque = EUnion ( T, header )
    type deqelt = EUnion ( T, member )

    type InResults = [ wasempty, wasnonempty ]
    type RemResults = [ wasempty, nowempty, nonempty ]

    proc Empty ( d: deque ) b: boolean = . . .
    proc InsertHead ( d: deque, e: deqelt ) r:InResults = . . .

    proc InsertTail ( d: deque, e: deqelt ) r:InResults = . . .

    proc RemoveHead ( d: deque, e:ref deqelt ) r:RemResults = . . .

    proc RemoveTail ( d: deque, e:ref deqelt ) r:RemResults = . . .

end module QueueDefinition
```

The module is generic in the type of value to be stored in elements of the deque.

Here are the details of the transfer request queue structures:

```
Type TRBTag = [disk, magtape, network, terminal ...]

Type TRBdata is record
    . . .
    case tag: TRBtag of
        disk: record
                var controller:DiskIndex
                var command
                var status
                end record
        magtape: . . .
                 . . .
    end case
    end record

var IOQueueDef: DequeDefinition[TRBdata]

type TRB = IOQueueDef.Deqelt
```

There is a variant of the TRBdata type for each type of device. The disk request block indicates which controller is to receive the request, as well as containing fields for the command to be given and the status information to be received.

IOQueueDef defines a particular kind of deque, whose elements have a value of type TRBdata. This deque is used throughout the module IO. In particular, the deque element type is renamed TRB.

Both IOPostQueue and IODoneQ8 . . IODoneQ11 are all implemented with deque's.

```
module IOPostQueue is
    acquires SoftInt4
    requires IOQueueDef, TRB
    provides IODone, SendIOPost, GetIOPost

    rename IODone = SoftInt4

    var IOPostQ: Deque

    proc SendIOPost ( C: TRB ) =
        InsertTail ( IOPostQ, C )
        raise IODone

    proc GetIOPost ( C: TRB ):RemResults =
        RemoveHead ( IOPostQ, C )
end module IOPostQueue

module IODoneQ11 is
    acquires SoftInt11
    requires DiskComplete, MagTapeComplete, . . .
    provides IODone11

    var DoneQ: Deque

    proc IODone11 ( C: TRB) =
        InsertTail ( DoneQ, C)
        Raise SoftInt11

    handler for SoftInt11 =
        var C: TRB
        while not RemoveHead( DoneQ, C ) = wasempty do
            case C.tag of
                Disk: DiskComplete ( C )
                MagTape: MagTapeComplete ( C )
                . . .
            end case
        end handler
end module IODoneQ11

module IODoneQ10 is . . .
module IODoneQ9 is . . .
module IODoneQ8 is . . .
```

IOPostQueue uses priority level 4, and signals IODone (SoftInt4) for each completion. Typically the procedure GetIOPost would be used inside a handler for IOdone, which would loop until the queue was empty. Because the hardware queue instructions are uninterruptible, no critical sections are needed to synchronize Send and Get.

IODoneQ11 maintains a queue of completed TRB's for disk, magnetic tape, and network controllers. Its handler drains the queue each time it is invoked, using the tag field of each TRB to determine which completion procedure to invoke. Again, no critical regions are needed. IODoneQ10 . . IODoneQ8 would have the same form, for different devices.

4.5.2. Disk Drivers

Recall that module *devices* defined a vector of hardware disk controllers. Each controller driver maintains a single sorted list of transfer requests. To simplify this example I have used a strict FIFO scheduling policy. Each driver maintains a flag indicating whether a command is in progress, and a variable containing the TRB being processed.

```
type Driver(I: DiskIndex) is
     acquires Devices.DiskController
     requires IODone11
     provides Submit, Complete

     var DiskQueue: Deque
     var Current: TRB
     var Busy: Boolean := False

     proc SelectCommand ( D: driver) =
          var C: TRB
          crit ( Busy, priority = SoftInt11.priority )
          if RemoveHead (DiskQueue, C) = wasempty
          then Busy := false
          else Current := C
               DiskController[I].Command := C.Command
          end crit

     handler for DiskController[I].condition is
          Current.status := DiskController[I].Status
          IODone11 ( Current )

     proc Complete ( D: driver ) =
          if << error >> then <<retry>> else
          SendIOPost ( Current )
          Selectcommand

     proc Submit ( D: driver, C: TRB) =
          InsertTail ( DiskQueue, C)
          crit ( Busy, priority = SoftInt11.priority )
          if not busy then
               Selectcommand
          end crit
end type driver
```

The driver synchronizes initiation, interrupt handling, and completion without using critical regions, by programming them such that only one can be underway at any one time:

- The procedure *SelectCommand* removes a TRB from the queue, records it, and starts the controller.
- When the controller interrupts the CPU, the handler copies the status register into the TRB, and moves it to IODoneQ11.
- IODoneQ11 passes the TRB to the procedure *Complete*, which either retries it (if it failed) or starts a new one.

However, critical regions *are* needed to protect the *Busy* flag for each driver. Since the interrupt handler does not lock *Busy*, the priority needed to protect *Busy* is that of the handler for *IODoneQ11*, namely 11.

To round out the module *DiskManager*, we declare a vector of drivers, and the routines *DiskSubmit* and *DiskComplete* to route TRB's to the appropriate drivers. *TransferRequest* is likewise straightforward.

```
        var Cont: array [DiskIndex] of ref driver
            init
            for I := DiskIndex do Cont[I] := new driver(I)
            end init
        proc DiskSubmit ( C: TRB ) =
            Submit ( Cont[C.controller], C )

        proc DiskComplete ( C: TRB ) =
            Complete ( Cont[C.controller] )
    end module Disk Manager

    module MagTapeManager is . . .
    module NetworkManager is . . .
    module TerminalManager is . . .

    proc TransferRequest ( R: TRB) =
        case R:TRBtag of
            . . .
        disk: DiskSubmit ( R )
            . . .
        end case
```

Having discussed the individual modules, we now juxtapose them in an abbreviated description of the entire module IO.

```
module IO is
    acquires loader.devices
    provides IOPostQueue,TransferRequest, TRB

    module IOPostQueue is
        acquires Softint4
        requires IOQueueDef, TRB
        provides IODone, SendIOPost, GetIOPost

        rename IODone = Softint4
        var IOPostQ: Deque
        proc SendIOPost ( C: TRB ) = . . .
        proc GetIOPost ( C: TRB ):RemResults = . . .
    end module IOPostQueue

    module IODoneQ11 is
        acquires Softint11
        requires DiskComplete, MagTapeComplete, . . .
        provides IODone11

        var DoneQ: Deque
        proc IODone11 ( C: TRB) = . . .
        handler for Softint11 = . . .
    end module IODoneQ11

    module IODoneQ10 is . . .
    module IODoneQ9 is . . .
    module IODoneQ8 is . . .

    Module DiskManager is
        requires IODoneQ11, SendIOPost
        provides DiskSubmit, DiskComplete

        type Driver(I: DiskIndex) is
            acquires Devices.DiskController
            requires IODone11
            provides Submit, Complete

            var DiskQueue: Deque
            var Current: TRB
            var Busy: Boolean := False
            proc SelectCommand ( D: driver) = . . .
            handler for DiskController[I].condition is
            proc Complete ( D: driver ) = . . .
            proc Submit ( D: driver, C: TRB) = . . .
        end type driver
        var Cont: array [DiskIndex] of ref driver
        proc DiskSubmit ( C: TRB ) = . . .
        proc DiskComplete ( C: TRB ) = . . .
    end module Disk Manager

    module MagTapeManager is . . .
    module NetworkManager is . . .
    module TerminalManager is . . .

    proc TransferRequest ( R: TRB) = . . .
end module IO
```

4.5.3. Reflection

The example from VAX/VMS actually says as much about when *not* to use explicit synchronization as about when to use it. The VAX architecture helps considerably in supporting interrupt handling; the **conditions** and **critical regions** simply imbed the obvious program structures in a strongly typed notation. Had the queue operations not been indivisible, they would have been placed in critical regions. In such situations, the priority computation techniques would be very helpful in assuring mutual exclusion. The critical region construct is a significant improvement over both Modula and Concurrent Pascal, in that it allows direct communication among various device drivers and interrupt routines, via dynamically nested critical regions.

Still, one might argue persuasively that the effort of putting a specialized feature into one's implementation language solely for this purpose, would not be economically justifiable. For any individual operating system, this might well be true. Nonetheless, modelling interrupts as conditions and deriving priorities from locks provides the formal basis for verifying system properties, even if the translation is done by hand. In section 7.4 we will discuss implementation techniques that might bring costs within acceptable limits for specific situations.

4.6. Summary

We have introduced hardware and software interrupt facilities into system descriptions as support for strongly type **conditions**. We have integrated the IPL register into the implementation language in support of **critical regions**, as a convenient special case of *data locks*. The proposed methodology contributed to this system design in the following ways:

- The device registers were declared in an environment having access to the hardware I/O memory, then **acquired** by their managers.
- The **acquires** construct allowed exclusive access by each device manager to its device, without unnecessary levels in the *contains* hierarchy.
- The language system specifies the *form* of interrupt modules, but allows the operating system to supply the *contents*.
- The language system can either translate critical regions into priorities, or verify that the indicated priorities are sufficient, at compile time.
- The interrupt vectors in the software control block can be initialized during system generation, rather than during startup, without requiring a central list of all handlers.

CHAPTER 5
BOOTSTRAPPING

This chapter describes the design of a bootstrapping mechanism for a DEC VAX-11 operating system, similar to the mechanism actually used in VAX/VMS. The design identifies the execution environments involved in the bootstrap sequence, and describes the mechanisms that connect them, to configure, load and start the operating system. The design is novel in the following ways:

- It describes segment layout using an abstract data type, *layout*.
- Each environment manager initiates its environment by signalling a *startup* condition.
- Each system module can place configuration code in the system generation environment.
- Address translation, primary memory allocation, and demand paging facilities conform to the same modularity and hierarchy in the bootstrap sequence as they do in the running system. In particular, the permanent environment will load and run correctly whether or not the demand paging facility is present.
- The "trick" used to turn on virtual memory mapping without abrupt discontinuity in control flow, is confined to a few lines of code in a startup condition module, where its (rather peculiar) connections to other system components can be identified.
- The mechanism for disposing of startup code is embedded in the startup condition's termination protocol, where its connections likewise can be scrutinized.

Although one purpose of the case study is to demonstrate that the methodology is effective for real systems, the reader would quickly tire of the bookkeeping details present in a complete implementation. Therefore, I give only the specifications of certain modules.

5.1. Segment Layout Description Language

To describe the paged environments of VAX, we shall need data abstraction to describe how segments are arranged in address spaces. Most linkers use some form of sequence or tree language to describe layout (see, for example, RSX-11 Overlay Description Language [RSX 78]). The language described here is sufficient to support the loading example that follows, and furthermore can be generalized to support intricate combinations of overlapping address spaces.

```
Layout ::=      Leaf |
                LayoutVariable |
                ( Layout {,Layout} )

Leaf ::=        Segment |
                AddressConstant |
                Pool

LayoutVariable ::= Identifier
```

Two leafs occurring consecutively in a layout are to occupy consecutive virtual addresses. The attributes of a segment are associated with the segment itself, not with the layout (or layouts) in which it appears. An address constant appearing in a layout specifies the endpoints of the leafs adjacent to it. A pool is a shorthand declaration for both a vector-of-storage and the segment in which it resides, where the size of the vector is determined by the layout in which it occurs. (A pool may appear in exactly one layout.)

Layout variables obey "object semantics" when they appear in other layouts, thus providing a mechanism by which two layouts may share sublists. Any individual layout may only contain one occurrence of any particular leaf, but the collection of layouts in a system form a forest, with common subtrees. Layout variables are also a *forward reference* mechanism: a layout appearing early in a system description can include a previously-declared layout variable, as a place-holder for segments that have not yet been declared.

This language definition is sufficient for the example of this chapter. More generally, however, a layout could be any directed, acyclic graph. Multiple occurrences of a segment only constrain the extent to which that segment may be inter-linked with other segments. In computer systems with instruction sets supporting PC-relative addressing, two segments may be linked relative to one another if they have a unique least common ancestor. (A common ancestor of a pair of leafs is least if none of its descendants are common ancestors of the pair.)

5.2. Startup Conditions

Exception handling mechanisms for startup require close cooperation between the operating system and the language system. The former must supply the execution environment in which the latter elaborates a procedure invocation. Here I shall propose a simple model for the interaction, sufficient to characterize the VAX solution, but not a detailed proposal for a real system.

Executing an exception handler is very much like executing an ordinary procedure call. The main differences are that the procedure address must be retrieved from a set of eligible handlers (rather than being known at compilation time), and that the procedure receives parameters from both the signalling and the enabling contexts. These differences might be implemented by additions to the procedure invocation protocol such as instructions to set up and tear down the interpretation stack frame. In general, one might expect the exception protocol to add instructions

1. Before transferring to the procedure
2. Just after transferring
3. Just before return
4. After return

In multi-environment systems the protocol might also effect a transfer to another execution environment. For example, a context switch operation in general can be decomposed into the steps:

- Save context
- Change "Current Context" register
- Load context

These steps would be carried out at handler entry, then reversed at handler exit.

Since each startup condition is (potentially) specialized to a particular environment, we shall define the protocol for each startup mechanism in the module that declares the condition, as a pair of parameterless procedures named *Init* and *Quit*.

For example, the condition handler to start up a bare machine would have to set up the execution stack pointer. If the handler finished executing without some ordinary process taking over the CPU, the handler might execute a "busy waiting" loop waiting for device interrupts. Here is the condition module:

```
Condition Module Startup is
    provides
        condition()
        const Priority = MinPriority
        proc <host> Set ( StartingPoint address )

    var ExecStack: Stack (MinDepth) of StorageLocation
    proc Init =
        CPU.StackPointer := ExecStack.TopPointer

    proc Quit =
        while true do { Blink console lights! }
```

One additional detail of startup handlers must be mentioned here. Since oftentimes a loader and the segment it loads reside on different storage devices, the starting address of the code being loaded is normally stored with the segment rather than with the loader. To avoid relinking, the starting address is usually stored at some known location (such as the first one!) in that segment. We shall refer to that location as the *transfer vector* of the segment, and treat it as a component field of the type *segment*.

5.3. Bootstrap environments for VAX/VMS

The major components of the VAX system address space are linked together by only a small number of pointers. This allows the size of many tables to be configured during bootstrapping, including the system page table itself! VAX/VMS accomplishes this by executing a substantial piece of system code before enabling memory mapping. That code interrogates the operator for changes in system parameters, then sets the sizes of the appropriate tables, allocates the system page table, reads in the kernel code and data, and enables mapping. Further initialization code, executed with mapping enabled, resides in a separate segment that is discarded afterwards. The distinct execution environments of VAX/VMS may be characterized as follows:

1. Memory configuration environment: ROM code in the memory controller arranges the mapping from physical addresses to working memory modules in a convenient pattern.

2. VMB environment: essentially a bare machine, except that the floppy disk drive is known to contain a directory-structured set of files. The program VMBoot constructs a bit map identifying the working pages of the physical address space, provides various device communication services to higher levels, sets up the SysBoot environment, and reads in its program segment from one of several permissible bootstrap devices.

3. SysBoot environment: like VMB, except that it presumes the bit map and device services. The SysBoot environment is used both by SysBoot (the VMS loader) itself and by various diagnostic programs. The program SysBoot configures the system tables, initializes the system page table and page frame management data structures, reads in executive code and data, enables memory mapping, and transfers control to the executive initialization code.

4. Permanent environment: those portions of the operating system that are permanently core resident, including process multiplexing, paging, swapping, and other facilities.

5. System environment: provides access to all system segments except the "throw away" segment. Contains no initialization code per se. Can be entered by trap, interrupt or Change Mode instruction while executing either on the interrupt stack or in a full process context.

6. Init environment: uses the interrupt stack, and virtual memory (system space only), including all of the memory initially addressable. The initialization code resides in a small "throw away" segment that is removed from the system page table when initialization is complete.

7. Multi-processing environment: large fixed-size set of process contexts, with full synchronization facilities. All system activities except interrupt and context swap code execute within processes, either by system call from user processes, or from dedicated system processes.

The program skeleton on the next page shows the relationships between environments 2 through 7. At this point we depart from the "real" VMS bootstrapping sequence and discuss instead my simplification of it. For example, the skeleton shows a *startup* condition for several environments, whereas VAX/VMS simply transfers control from one loader to the next. In preparing this case study I have taken care to confront each step of VAX/VMS bootstrapping, to see whether it could be incorporated into my redesign. The abstraction presented here includes the major design obstacles I uncovered, but suppresses many of the ordinary chores.

```
environment module VMBEnv is
    requires HostEnv, linker, VAX.Devices
    provides    var AdSpc: AddressSpace, var PR: ProgramRegion
                condition module VmbStart, proc<host> download
```

```
Module VMBoot
    requires HostEnv, VMBEnv
    provides device drivers, var PfnBits: MemoryPageAllocationBitstring

        environment module SysBootEnv
            requires VmbEnv
            provides    var AdSpc: AddressSpace, var PR: ProgramRegion
                        condition module SysBootStart

Module SysBoot
    requires HostEnv, SysBootEnv, VAX.Devices
    provides
        module SptManager
            provides    type PermSeg, var SptSeg: PermSeg
                        var SptMap: layout, var Spt: PageTable
                        proc InitSptAddressSpace
        var SysPerm, IntStkSeg, ScbSeg, IOSeg, LockedPoolSeg, GPTSeg: PermSeg

        environment module PermEnv
            provides    var AdSpc: Address Space :=
                            (SptManager.Map, { permanent segments })
                        var PR: ProgramRegion
                                := { ... subset of permanent segments ... }

        type PagedSeg, var PfnSeg, SysPhdSeg: PermSeg
        var SysSwapSeg, ThrowSeg, PagedPool: PagedSeg
        var SysSegPoolA, BalanceSlots:layout

        environment module SysEnv
            provides    var AdSpc: AddressSpace := (SptManager.Map,
                                { all segments except ThrowSeg } )
                        var PR: ProgramRegion = { all available segments }

        environment module InitEnv
            provides    var AdSpc: AddressSpace :=
                            ( SptManager.Map, { all initial segments })
                        var PR: ProgramRegion = { ThrowSeg }
                        proc ThrowAway
                        condition module SysInit

        handler<SysBootEnv> for SysBootStart =
            -- Build initial segment list
            -- Configure LoadSized segments
            InitSptAddressSpace
            -- raise SysInit
module ProcessManager
    requires SptManager, HostEnv, SysEnv, PermEnv
    provides

        environment type Process
```

This overview shows three significant features that we shall investigate:

- Three environments supply *startup* conditions that form the bootstrapping chain.
- The three mapped environments, PermEnv, SysEnv, and InitEnv, all originate in the SysBoot module, which both constructs the necessary data structures initially and supplies the routines and access functions needed to manipulate them during normal operation.
- The system page table and the permanent environment are introduced without reference to demand paging, even though the permanent and paged segments all reside in the same system page table.

5.3.1. General properties of the loaders

Each loader in the system is responsible for accumulating a set of segments on the host machine, identifying a starting address within that segment set, transmitting the segments to the target machine, loading them into the execution environment, and transferring to the starting address. Each loader uses the module *Linker* to implement the types *Segment*, *Layout*, and *AddressSpace*. We shall assume that the type *segment* results in a token that can be used both on the host machine and the target machine to locate the segment on the bootstrapping disks.

```
Module linker
    provides
        type SegmentAttribute = Absolute,Volatile,LinkSized,
                                LoadSized,Uninitialized
        type SegAttrList = set of SegmentAttribute
        type segment( attr:  SegAttrList )
        type layout
        type AddressSpace is record
            segments: segmentset
            Map: layout
```

With these types, a loader can use a layout to determine the base addresses of a set of segments, cause them to be linked, download them to the bootstrap medium, and load them on the target machine. Once in place, the loader uses a startup condition (implemented with a transfer vector) to initiate execution of the program it has loaded. The handler for that condition would be the next loader in the bootstrapping sequence. Thus the primitive bootstrap function provided by the VAX "intelligent console" loads VMBoot into VMBEnv and signals *VmbStart*. VMBoot loads SysBoot into SysBootEnv and signals *SysBootStart*. SysBoot constructs the system page table, creates the configured system tables, loads the operating system code and data segments, and signals *SysInit*. To illustrate these concepts, here is a more detailed description of the module VMBoot

```
Module VMBoot
     requires VMBEnv
     provides device drivers, SysBootEnv, SysBootStart, PfnBits, MemoryPool

     var<VmbEnv> PfnBits: bitstring[0 .. PhysicalCoreMax-1]

     environment module SysBootEnv
          requires VmbEnv
          provides AdSpc, PR, MemoryPool

          var Program: segment
          var MemoryPool: pool
          var AdSpc: AddressSpace
               AdSpc.Map = (VmbEnv.AdSpc.Map, Program,
                                   MemoryPool, [PhysicalCoreMax]
               AdSpc.Segments = Leafs(AdSpc.Map)
          var PR: ProgramRegion := ( Program )

          condition module SysBootStart is
               requires Program
               provides condition, priority

               condition <SysBootEnv>()  -- handlers must reside in SysBootEnv
               proc<host> Set ( startingpoint: address ) =
                    Program.TransferVector := startingpoint
          end module SysBootStart
     end module SysBootEnv

     proc<host> DownLoad =
          -- Link Program and MemoryPool using AdSpc.Map
          -- Write Program to disk
          VmbEnv.Download

     handler<VmbEnv> for VmbStart =
          -- Fill in PFNbits, checking pages for errors
          -- load SysBootEnv.Program into its base address
          AbsoluteGoTo(Program.TransferVector)
```

VMBoot implements the environment SysBootEnv, providing it with an address space, a program segment, a storage pool, and a startup condition. The downloading procedure links the program segment and places it on the boot disk. The loading procedure is implemented as the handler for VmbStart. During translation some subsequent module enables a handler for SysBootStart, which causes the language system to call SysBootStart.Set, recording the starting address in the transfer vector. After the loader reads in the program segment, it can fetch and go to the starting address.

5.3.2. Modularity vs. Bootstrapping

The SysBoot procedure must carry out numerous bookkeeping details on behalf of several different modules of the operating system. Therefore, we must consider it to be an integral part of the operating system, rather than a separate module. In the proposed structure, the SysBoot module implements the system page table, the page frame allocation module, the demand paging mechanism for paged system segments, and all of the storage allocation zones in the system virtual address

space. Code to initialize these structures resides in the SysBoot environment, whereas code for conventional manipulation of these structures resides in the three mapped system environments.

Although not part of this design, if subsequent system levels needed to perform initialization in the SysBoot environment, the SysBoot module could define another condition in that environment, and signal it from within the handler for SysBootStart.

5.4. Address Translation, Page Frame Allocation, and Demand Paging

These three services interact closely in VAX/VMS, as in most operating systems. Inter dependencies among them can foil attempts to maintain a hierarchical system structure. The interaction is especially apparent during bootstrapping, where

- page frames are allocated to contain the address translation and demand paging facilities
- address translation is set up for the page frame allocation and demand paging facilities
- the page frame management and address translation tables are filled with information needed by the page fault handler.

VAX/VMS keeps the mechanisms reasonably cleanly separated, but makes no claim of hierarchy. The description I propose generally reflects the VAX/VMS structure, but also admits the possibility of hierarchy. (A detailed design would be beyond the scope of this chapter.)

5.4.1. Table Sizes

In VAX/VMS the sizes of the address translation table and page frame management table depend upon one another. Both tables occupy physical page frames (affecting the page frame management table) and virtual pages (affecting the system page table). A good system design would make the system page table *exactly* big enough to translate the system address space, and make the page frame table *exactly* big enough to keep track of those page frames that are *not* occupied by the resident monitor. We shall see how VMS approximates this ideal, and how it could be accomplished while still maintaining separation between the two modules.

The size of a VAX page table is 1/128th of the size of the address space it is mapping. That address space contains four classes of segments:

1. System page table (SPT)
2. Page frame table (PFT)
3. other Permanent Segments (PS)
4. Non-Permanent segments (NPS)

Using the above acronyms loosely to refer either to the segments themselves or to the sizes of the segments, we can write the formula for SPT as

$$SPT = \lceil (PS + NPS + PFT + SPT) / 128 \rceil$$

The VAX page frame table contains 18 bytes for every (512 byte) page of dynamically allocated storage. That storage is the amount of primary memory left over when all of the permanent segments have been allocated, so

$$PFT = \lceil (CoreMax - PS - SPT - PFT) * 18 / 512 \rceil$$

The first recurrence relation can be resolved as

$$SPT = \lceil (PS + NPS + PFT) / 127 \rceil$$

but the second one is more difficult. Rather than an analytic solution, DEC chose a conservative method, namely computing the size of the PFT before expanding the SPT to accommodate the PFT itself. However, a more exact solution may be obtained using a polynomial root-finding method, such as bisection:

```
Proc PreSetPft =
     { CoreMax, PS, NPS, PFT, SPT in units of one page }
     Function NewSpt = ( PS + NPS + PFT + 126 ) DIV 127
     Function NewPft = ( ( CoreMax - PS - SPT ) * 18 + 529 ) DIV 530
     PFT := 0
     SPT := NewSpt
     PFT := High := NewPft     -- Biggest possible PFT
     SPT := NewSpt
     PFT := Low  := NewPft     -- Smallest possible PFT
     While Low < High do begin
          PFT := Mid := (Low + High) DIV 2
          SPT := NewSpt
          If ( Coremax - PS - SPT - PFT ) > ( PFT * 512 ) DIV 18 then
               PFT := Low := Mid + 1
          else High := Mid
          end
```

The initial value for *High*, above, is the size for PFT used by DEC. The difference between High and Low initially could be as great as 10 pages when VAX becomes available with a gigabyte of physical memory, but for presently available systems the difference would be at most one page, and there might be no difference at all.

Nonetheless, to preserve separation between the address translation and page frame management modules, we would like to conceal the amount of storage overhead each module introduces. Let the SptManager provide a function whose value is the difference between the size of the physical memory and the combined sizes of the permanently resident segments. This information can be computed before actually allocating space for those segments, as long as the individual sizes have been specified. Should the size of any segment be changed (e.g. the PFT segment!), the value of that function would change correspondingly.

The bisection procedure would look like this:

```
Procedure SetSize ( S: segment; size: integer )  { provided by SPT manager }
Function FreeSpace: integer                      { private to SPT manager  }
Proc PreSetPft =
     Function NewPft = ( ( FreeSpace + PFT ) * 18 + 529 ) DIV 530
     SetSize(PFTseg, 0)
     High := NewPft -- Biggest possible PFT
     SetSize( PFTseg, High )     -- VAX/VMS stops here
     Low  := NewPft      -- Smallest possible PFT
     While Low < High do begin
          Mid := (Low + High) DIV 2
          SetSize(PFTseg, Mid)
          SPT := NewSpt
          If FreeSpace > ( Mid * 512 ) DIV 18 then
               Low := Mid + 1
          else High := Mid
          end
     SetSize(PFTseg,High)
```

By this technique we can accommodate the discrete jumps in the size of the SPT without making the PFT manager aware of its implementation. Should the system page table be redesigned, no further changes to the PFT manager would be needed. The conservative solution used by DEC is embodied in the first three steps of the program above.

5.4.2. Keeping Demand Paging Separate

Having disentangled the size of the SPT from the size of the PFT, we can consider how to keep the demand paging data separate from the address translation and storage allocation information. Fortunately, data abstraction provides a straightforward solution. Both the SPT and PFT are vectors of records, where part of the contents of each record is solely for the paging mechanism, and of no interest to the table manager (except for its size). Therefore, we define two types, *SectionTablePointer* and *PfnData*, which are *required* by the SPT and PFT modules, respectively, but only so that they can be embedded in records. The SPT manager embeds the section table pointer in a variant field of a page table entry, indicating which segment contains the missing page. The page frame table manager embeds the PfnData in a dynamic record type, providing a pool of such records (1 per page frame), with facilities for allocation, deallocation, and building doubly circularly linked lists.

```
type SectionTablePointer is forward  -- only the size is needed by PTE

type PageTableEntry is
    . . .
    case valid: boolean of
        true: . . .
        false: STP: SectionTablePointer

type PageTable( s: size ) = array[0 .. s-1] of PageTableEntry

Module SptManager
    requires PageTable, PageTableEntry, PfnBitsManager
    provides
        type PermSeg
        var SptSeg :PermSeg({LoadSized, Uninitialized})
        var<SptSeg> Spt: PageTable

Type PfnData is forward

module PfnDataBase
    requires PfnBitsManager, SptManager, PfnData
    provides
        var PfnSeg: PermSeg ( {LoadSized,Uninitialized} )
        type PfIndex
        type PfnDescriptor is record
            succ, pred: PfIndex
            data: PfnData
        var PfnTable: array[PFIndex] of PfnDescriptor
        type PfnList: -- linked list of PfnDescriptor
        proc AllocPfn( Size: integer ):PfnList
        proc Free ( L: PfnList )

module PagedSegManager
    requires PfnDataBase, SptManager, SegDesc
    provides
        type PagedSeg
        var SysPhdSeg: PermSeg (LoadSized, Uninitialized)
        type SectionTablePointer
        type PfnData
        type SectionTable(var PT:PageTable)
        var<SysPhdSeg> SysSectionTable: SectionTable(Spt)
```

With this data organization, the permanently resident segments are protected from being swapped, in two ways:

1. The paged segment manager does not have access to their segment descriptors (indeed, their descriptors are not addressable in any of the mapped environments.)

2. The pages that the permanent segments occupy are excluded from the PfnDataBase.

When a page fault occurs, the paging module can examine the SPT to find the entry for the faulting page, and use its SectionTablePointer to find the paged segment in which the page resides. The page replacement algorithm would select from lists of occupied page frames, all of which would by definition come from the PfnDataBase. Therefore, the paged segment manager would never swap out a page from a permanent segment. (Some page replacement schemes periodically scan the entire page table, collecting usage information from every entry. Such a procedure would

use the page frame number in the page table entry as an index into the PfnTable, discarding information on pages beyond the bounds of the table.)

5.4.3. Putting the pieces together

The modules as described above obey a total ordering according to the *uses* relation, except for the two forward type definitions. (Even these could as well have been generic parameters to the modules.) The SPT manager provides the system page table, the ability to create permanently resident segments, and a pool of leftover space. The PfnDataBase resides in a permanent segment, and converts the leftover space into a pool of page frames. The PagedSegment manager uses the SPT and page frame pool to implement demand paging. To complete the analysis, however, we must examine the startup sequence, to see whether the design is feasible, and to see whether the modularity is preserved.

The sysboot loader must perform the following tasks:

1. Set the sizes of configured segments
2. Allocate storage for permanent segments
3. Build the system page table
4. Initialize the Pfn pool
5. Allocate storage for paged segments
6. Load the code and data segments from disk

To accomplish them in an orderly fashion, we group them according to the modules they affect, and find that both modularity and hierarchy can be preserved.

The system page table, which is lowest in the hierarchy of the three modules considered, must be initialized after all other segment sizes are set, and before the Pfn pool can be constructed. Therefore, the SPT manager supplies an initialization procedure that computes the size of the SPT, allocates space for the SPT segment and all other permanent segments, and fills in their page table entries:

```
Proc StartSpt
        {precondition:  all segment sizes other than SptSeg
                have been set}
        Determine SPT size from SptManager.Map
        SetSize(SPTseg, computed value)
        Allocate primary storage for all permanent segments
        In SptSeg, construct system page table entries
                for all permanent segments
```

Initializing the PFN pool involves activities both before and after setting up the SPT. The size of the PFN segment must be set before the system page table can be built; after setting up the SPT, the PFN segment is available to contain the page frame descriptor pool:

```
Proc StartPfn
        {precondition: all segment sizes other than PfnSeg and SptSeg
                have been fixed}
        PreSetPfn
        StartSpt
        InitPfnPool
```

To start up the paged segment manager, we must start up the PfnDataBase, then allocate space and fill in page descriptors for paged segments. The positions of those segments in the SPT are defined by SptManager.Map.

```
Proc StartPagedSegManager
        {precondition: same as for StartPfn}
        StartPfn
        AllocPagedSegs
```

Finally, we can describe the "main program" of the SysBoot loader:

```
handler <SysBootEnv> for SysBootStart =
        Initialize segment sizes for configured segments
        StartPagedSegmentManager
        Load code and data segments
        Signal startup of the mapped environments
```

Although control flow passes between modules several times during startup, the startup code conforms to the same modularity as the running system. Furthermore, the *calls* relation during startup totally orders these three modules, suggesting that higher modules could be removed and the lower modules would still start and run correctly.

5.5. Starting Up The Mapped Environments

As with most virtual memory architectures, there is no elegance whatsoever in the manner in which VAX/VMS first enables memory mapping. However, one would like to encapsulate the peculiar act in a small portion of the system, where it can be understood long enough to debug, and then can be ignored. First we shall discuss the trick that VAX/VMS uses, then see how it can be encapsulated in a *startup* condition.

5.5.1. How it works

Enabling memory mapping normally causes a discontinuity in control flow, because the CPU suddenly begins interpreting the addresses in the program counter, stack pointer, and other registers as virtual rather than physical addresses. Normally the new interpretation is not a valid one, unless the designer chooses carefully what the initial memory mapping will be. An identity mapping, for example, assures continuity.

On a PDP-11 an identity mapping, in low core, is convenient, because the interrupt and trap vectors appear at predefined, low addresses which are virtual if mapping is enabled, or physical if not. However, the VAX architecture requires the operating system to occupy addresses in the upper half of the virtual address space, which

are all greater than the largest physical address. Therefore, a simple identity map will not solve the problem. Instead, VAX/VMS brings about the transfer in two steps, involving three address spaces. The VAX address mapping architecture uses three page tables: a system page table for translating addresses in the upper half of the virtual address space, and a pair of page tables for translating addresses in the lower two quarters of the virtual address space. The system page table is shared by all processes, whereas there is a separate pair of "user page tables" for each process. The initialization procedure is present simultaneously in the sysboot environment's unmapped address space, in a process-private address space, and in the system address space. Sysboot constructs the process-private page table to supply an identity map for the "Map enable" instruction, removing the original discontinuity. The instruction following "Map enable" is an unconditional jump to the very next instruction -- but using its address in the system address space. The three instructions, and the address spaces in which they execute, are as follows:

```
        CPU.MapEnable := true     -- Physical address space
        Go To A                   -- process private address space
    A:  Continue                  -- system address space
```

This is the heart of the "trick", but there is more: The process-private page table is constructed by "equivalencing" the system page table! Since the initialization procedure is present in the system address space, the page containing it is described in the system page table. VMS looks up the physical page frame number of the page containing the the startup procedure, and selects a subrange of the system page table to be the process page table, such that the entry describing the relevant page will occupy the same virtual and physical pages. (This implies that many other system pages are also twice-mapped, but the process-private page table is only used to execute one instruction.)

5.5.2. Relating the "trick" to Environments

We shall embed the map enabling trick in *SysInit*, the startup condition for the mapped environments. The condition's *set* procedure records the *logical* address of the handler. The *signal* procedure, invoked in the SysBoot environment, determines the virtual address of the handler using SptManger.Map, looks up its physical address, sets up the process-private page table, and transfers control to the handler by jumping to its *physical* address.

To insert the "clever" instructions into the handler code, the condition module defines a handler initiation procedure, which the language system expands in line at the beginning of the handler. This code sets up interrupt stack pointer and other environment features, enables mapping, jumps to the system address space, and then invalidates the process-private page table.

```
condition module SysInit is
    requires SptManager
    provides condition, priority, signal

    condition <InitEnv>()  -- handlers must reside in InitEnv
    var<SysBootEnv> InitAddress:  Logical Address

    Proc<host> set ( LA: LogicalAddress ) =
        InitAddress := LA

    Proc<sysbootenv> Signal =
        X := VirtualAddress(SptManager.Map,InitAddress)
        Y := Translate(SPT, X)
        set up private page table
        AbsoluteGoTo(Y)

    proc Init =
        -- set up stack pointer, et cetera
        CPU.MAPEN := TRUE
        AbsoluteGoTo(A)
      A: -- private page table length := 0

end module SysInit
```

5.6. Disposing of startup code cleanly

Because startup code can be quite lengthy, and is executed only once, the virtual and physical memory it occupies can and should be "recycled" when startup is *complete. Ideally, there should be no trace of the startup code remaining when* startup is complete, but this can be difficult. Like jumping in a hole and pulling the dirt in after you, it is hard to dispose of the code that disposes of the startup code, without creating more code needing disposal.

One common technique for doing so places the startup code into a free storage pool without notifying the storage manager that the space is in use. When control transfers to the newly loaded system, the storage manager will "believe" that the storage is unoccupied. VAX/VMS uses this technique twice: to recycle memory from the unmapped environments, and to get rid of the program region for *InitEnv, which contains all of the once-only initialization code for the kernel* system.

The code and data of VmbEnv and SysBootEnv reside in the lowest pages of physical memory, which eventually become part of the PFN pool. During SysBoot, physical pages are allocated for permanent segments from the high end of physical memory first, leaving the low end undisturbed. When the PFN manager collects the unused storage to construct the PFN pool, it record the low-end pages as free even though they are still in use, but places them at the end of the free list so that they will not be allocated until much later, after mapping is enabled.

The program segment for InitEnv is a paged segment, rather than a permanent one,

so that it can be deleted and its pages reused. When the startup handler has finished its tasks and is ready to signal the process dispatcher, it must first delete its own program segment. The code to do so is written as "position independent code"; the startup handler copies the code into a storage pool intended for device handlers, then jumps to it. The code deletes the segment, then jumps to the process dispatcher.

I have not yet designed a representation for the InitEnv disposal mechanism. However, it could be embedded in the handler termination protocol for the *Sys/nit* condition. Specifically, the startup handler would signal the process dispatcher via the software interrupt mechanism, then terminate. After disposing of the startup segment, the termination protocol would execute a "return from interrupt" instruction, triggering the process dispatcher interrupt routine.

5.7. Summary

The proposed methodology has provided a useful conceptual framework for relating the many facets of bootstrapping in the VAX/VMS operating system. Startup conditions allow control to flow from lower system levels to higher level without violating hierarchy, and provide a logical place to connect the map enabling code and "disposal" code. Multi-environment modules integrate bootstrapping operations with conventional operations on page tables and storage pools. Incorporating the host environment in the system description allows operations on segments as bona fide data objects, making clear the relationships between address translation, storage allocation, and demand paging.

The design ideas presented here cannot be fully explored without actually building a language system and an operating system. However, each step in the bootstrapping phase of VAX/VMS has a place in the proposed methodology, and a niche in the resulting system description.

CHAPTER 6
HIERARCHY IN OPERATING SYSTEMS

In this chapter we shall identify dependencies among several VAX/VMS modules brought about by composition, procedure call, environment support, and data access, and verify that the union of these dependency relations defines a partial ordering of the modules involved. To do so we shall recast the relevant portions in a strongly typed notation, and in a way that was probably not envisioned by the system designers. Nonetheless, the redesign does not change the executable representation in any material way; only the source-language structure is different.

This study depends heavily on the study in the previous chapter. There we identified two relevant execution environments, *SysEnv* and *PermEnv*, which both used the System Page Table to implement their address translation. In fact, the logical address space of *PermEnv* is a subset of the logical address space of *SysEnv*. We shall use the fact that code executing in *PermEnv* can never cause a page fault, to help prevent a potential cycle in the module dependency graph.[6] As with other examples, this one simplifies certain details that would greatly complicate the description without posing any new technical problems.

6.1. Overview

The process management facilities in VMS consist of the following modules:

- Page Frame Manager: keeps track of the contents of all primary memory pages
- Paged Segment Manager: keeps track of all the pages associated with pageable segments, moving them in and out of memory on command
- Dispatcher: multiplexes the processor among ready processes
- Scheduler: maintains a queue of processes for each possible process state
- Pager: manages all user page tables
- Swapper: moves processes in and out of primary memory

---

[6] Because these two environments are already defined, this case study can be much briefer than the other two.

Some of the interesting relations among modules form partial or total orderings. When they do, they follow the order given above. The *calls* and *implements environment of* relations both conform to that ordering.

However, the data structures themselves seem to imply an ordering quite opposite to the others. The Dispatcher and Scheduler manipulate process descriptors, which include page tables that are themselves paged. Furthermore, the process descriptors themselves may be swapped by the Swapper. These dependencies contradict the *calls* and *environment* ordering, causing the overall system structure to degenerate into an unordered digraph.

By careful use of generic module definitions, we shall structure the data in such a way that the dependencies involving data structures conform to the same hierarchy as the other relations. Intuitively the ordering ought to exist, because the dispatcher and scheduler never access non-permanent memory, and the pager never accesses a non-resident page table.

6.2. The Data Structures

A process in VAX/VMS is rooted in a *software process control block* (Software PCB). This PCB contains a pointer to a *process header* (PHD). The process header contains the *hardware process control block* (Hardware PCB), working set and segment table information, and the process-private page tables (P0 and P1). A boolean variable in the software PCB indicates whether the process is executable (i.e. whether PHD is in core). Should the process be swapped out, the PHD pointer is replaced by a pointer to the process image in the swapping file.

6.3. The Page Frame Manager and Paged Segment Manager

These two modules have already been discussed in Chapter 5. Briefly, the Page Frame Manager allocates and deallocates primary memory pages, and provides a page descriptor facility for use by the Paged Segment Manager. The paged segment manager defines the Section Table data type, which keeps track of all of the pages associated with a given page table. Its representation includes a *working set*, which lists the incore pages in a form convenient for page replacement algorithms.

In Chapter 5 we defined the paged segment manager to also declare one particular section table, to accompany the system page table. In this chapter we will declare a section table for each incore process.

6.4. The Scheduler and Dispatcher

The scheduler and dispatcher incorporate the concept that some processes might not be executable, but do not access the information that is stored to describe a non-executable process. The scheduler provides the types SoftPCB and Phd, and preserves the property that an Executable SoftPcb contains a pointer to a valid Phd.

To reduce the complexity of this example, we shall suppress details of the Dispatcher inside the Scheduler. The Scheduler provides the following services:

- The type SoftPcb
- The type StateQueue, a circular list of SoftPcb's
- A particular state queue, the ReadyQueue
- Processor multiplexing
- The current process's SoftPcb

The scheduler resides in PermEnv, as does all the data that it accesses. One might wonder how the scheduler avoids accessing the user page table, which could cause a page fault. The scheduler's responsibility concerning these tables is limited to multiplexing the processor among ready processes. Only the executing process itself accesses the user page tables.

```
Type ProcessStateType = (Ready, IOWait, PageWait, SwapWait, ... )

Type <PermEnv> SchedulerType[SwapDataType, PageDataType: Type] is
     requires ProcessStateType, HardPcb
     provides
          type PHD is record
               HPCB: HardPcb
               PageData: PageDataType

          type SoftPcb is record
               prev,next: ^ SoftPcb -- private
               state: ProcessStateType
               case Executable: boolean of     -- ReadOnly
                     true: (exstate: ^ Phd<PermEnv>)     -- ReadOnly
                    false: (swapdata: SwapDataType)
               accounting: . . .

          Proc MakeExecutable( Proc: SoftPcb, StateVector: ^Phd )
          VProc MakeNonExec( Proc: SoftPcb ):^Phd

          function CurProc: SoftPcb -- current process

          type <PermEnv> StateQueue is queue of SoftPcb
          proc ChangeState(Old,New:StateQueue)

          var ReadyQueue: StateQueue     -- examine but not change
          proc MakeReady ( Proc: SoftPcb, OldState: statequeue)
          proc MakeWaiting ( Proc: SoftPcb, NewState: statequeue)
```

6.5. The Pager

The pager defines the notion of a pageable process. It supplies paging data to the Phd, and supplies page tables for the HPCB. It creates a vector of such pageable processes.

The page tables for user processes are themselves demand paged. Because the pager accesses these tables, it might be considered to reside in SysEnv rather than in PermEnv. Nonetheless, it cannot afford to incur a page fault while accessing such a page, because this would cause a recursive invocation of the page fault handler. To prevent this, the pager always "locks in core" any page table page it must access. To represent this concept, one could either distribute the pager across both PermEnv and SysEnv such that only a few key procedures were in SysEnv; or, one could redefine PermEnv such that its logical address space changed dynamically. For this example, however, we shall not page the user page tables.

The pager supplies the page fault handler for the hardware condition *translation error*. It examines the faulting address to decide which address space caused the fault, and calls the appropriate routine. That routine, if necessary, signals the condition "page not in core", which the page fault handler responds to by blocking the current process.

```
Type<PermEnv> PagerType[ SwapDataType: type] ( Poolsize: integer ) is
     requires SchedulerType, SectionTable
     acquires PageFault          -- condition signalled by hardware
     provides
          var scheduler: SchedulerType[SwapDataType, SectionTable]
          var PhdPool:  array [ 0..Poolsize-1 ] of record
               Header: Phd
               P0PT: page table
               P1PT: page table
               end
          proc HandleFault ( addr: address )

          handler for PageFault =
               if address in system space
                    then call PagedSegManager.HandleFault(address)
               else call HandleFault(address)
               endif [ on PageNotInCore:
                              MakeWaiting( CurProc, PageFaultQueue ) ]
```

6.6. The Swapper

The swapper defines the notion of a swappable process. It defines a swapped out process to consist of the Phd and all of the pages in the process's working set. When it swaps out a process it first makes the process non-executable, then copies out the Phd and pages, then returns the Phd to the pager's pool of Phds. The swapper code and data reside in PermEnv.

```
module <PermEnv> swapper is
    requires SchedulerType, PagerType, PermEnv, SysEnv,
             BalanceSetSize, MaxProcs, PfnDataBase.FreeListSize
    acquires PageFault
    provides
        type SwapData is private
        var Pager: PagerType [ SwapData ] ( BalanceSetSize )
        var ProcessPool:  array [ 0..MaxProcs-1 ] of SoftPcb
        proc Balance    -- analyzes process mix and swaps accordingly
```

6.7. Analysis

We wish to derive the dependency relation over these modules and the modules defined in Chapter 5. Because we have formulated environments as explicit program elements, we can obtain that relation from the program text directly. The domain of the relation is the set

{ *PermEnv, PfnDataBase, PagedSegManager, SchedulerType,*
PagerType, Swapper}

Each identifier required by a module connects it to the module providing that identifier.

Requiring Module	**Modules it depends upon**
PermEnv	none
PfnDataBase	PermEnv
PagedSegManager	PermEnv PfnDataBase
SchedulerType	PermEnv
PagerType	PermEnv PfnDataBase PagedSegManager SchedulerType
Swapper	PermEnv PfnDataBase SchedulerType PagerType

From this we conclude that the modules implementing storage allocation, scheduling, demand paging, and swapping are totally ordered with respect to one another.

6.8. Summary

This example is very short because the SysBoot module provided a permanent environment which did not depend in any way on paging. The chicken-and-egg problem in earlier systems came about because of inadequate distinction between address translation and paging. By making that distinction I have broken the dependency cycle; by making the distinction explicit in the program text the module ordering can be derived directly from the *provides* and *requires* clauses.

The ordering among these modules is highly desirable during system development, because it allows debugging of lower system levels without the complications of demand paging and swapping. It is desirable for formal analysis of system properties, because it allows the designer to isolate the one place where recursive page faults might occur, namely where the pager accesses a user page table. It is desirable for development of parallel versions, because it allows specialization for systems that do not require swapping and/or demand paging.

A subtle point about using generic modules: the scheduler is written to operate correctly regardless of the number and arrangement of SoftPcb's and Phd's. The pager defines the number of Phd's; the swapper defines the number of SoftPcb's. This approach differs from Janson's and Reed's approaches in Multics in that it views the type definition as more fundamental than the number of instances and how they are instantiated.

CHAPTER 7
EVALUATION

In chapter 1 we defined a set of criteria by which to evaluate the goodness of a methodology. Now that we have developed the methodology and applied it to several problems, we can use those criteria to evaluate the proposal.

7.1. Utility

Does the proposed methodology facilitate operating system design and construction? In each of the problem areas we found that the methodology contributed significantly to the solution of the problem:

- *Device communication:* By integrating the language system with the operating system, we were able to model interrupts as exceptional conditions. This allowed more flexible synchronization schemes than were possible in Modula or Concurrent Pascal. Controlling access to the I/O segment by means of environment *descriptions allowed explicit* control over access to individual devices.

- *System Integration:* By incorporating the host and unmapped target environments into the overall system description, we were able to combine system generation, linking, bootstrapping, initialization, and startup activities into a single, comprehensive system description. This allows automatic propagation of changes across environment management levels, static analysis of the bootstrapping sequence, better understanding of the relationship between initialization and startup, coordination of static and dynamic memory allocation, and a host of other consistency checks that are difficult on a piecemeal description.

- *Hierarchy:* By distinguishing address translation from demand paging, and by distinguishing data type definition from instantiation, we were able to impose and abstract structure on the VAX/VMS process management facility that exhibited a total ordering over the dependency relation. By recording environment dependencies explicitly in the program text, we were able to verify that the program actually obeyed the claimed ordering.

7.2. Clarity and Fitness

Does the methodology and its notation significantly increase the clarity and understandability of operating system descriptions? Is the meaning of the notation itself well understood? Is the notation appropriate to the problem domain?

Clarity and fitness contributed to the success of each case study. Condition modules, made possible by the integrated language/system approach, provided both a model and a notation for interrupt routines, that faithfully reflected their usual role in systems, and facilitated synchronization. Incorporating host and bootstrapping environments into system descriptions clarified many connections between system components that previously were defined obscurely through *ad hoc* system generation and initialization programs. Furthermore, it brought data abstraction tools to the problems of integration, boostrapping, and startup. The distinction between PermEnv and SysEnv clarified the dependencies between the pager and other system code, resulting in a demonstrably hierarchical structure.

The significant notations introduced in this thesis are:

- *A syntax for types and modules.* This notation was just one more variation on a set of well understood language features.
- *Environment modules.* These are ordinary modules that must supply certain specified features. The specifications themselves are ordinary program specifications. Making environment modules explicit entities fits the problem domain, because an operating system designer thinks in term of explicit domains.
- *Acquires clause.* The syntax and semantics of the clause are directly analogous to the conventional *requires* clause, with the additional constraint of exclusive access. The notation was introduced specifically to fit that concept, which appears commonly in system designs.
- *Environment annotations on program units.* This notation requires more justification than others, because it embodies the most novel aspect of the methodology. The notation highlights the fact that a program unit is an instance of an abstract type, whether the unit embodies data, control, or both. The environment annotation is in the form of an instantiation parameter, specifying the resource pool and type manager that implement the program unit. I have not undertaken a formal specification of the *meaning* of the annotation. However, one can conceptualize it by distinguishing between the *abstract meaning* of the program unit, which is not affected by the annotation, and the *executability* of the program, which must be derived from the environment information. In section 3.5.5, we discussed the notation in more detail, concluding that it was clear, fit, flexible, and terse. These conclusions were confirmed in the bootstrapping case study, where we found it suitable for describing the bootstrapping environments.

Another, less important notation was also introduced:

- *Condition modules.* For this feature we used the specification technique used in Euclid to specify *storage zone modules*, and followed the semantics proposed by Levin. We found the condition modules suitable for describing interrupts, including synchronization, and startup.

To summarize, each of the notations introduced is similar in form and meaning to some existing, well accepted notation. The notations and methods together produce comprehensive system descriptions that clarify important structural properties of systems.

7.3. Flexibility

Does the methodology facilitate system construction without unduly constraining system design? Facility and transparency have shaped the methodology at every step. Execution environments are defined in a way that accomodates both suspicious and cooperative environments. The operating system and its implementation language are integrated in a way that allows the language to facilitate use of the system's resources while removing only that transparency which the system design dictates be removed. The environment binding notation accomodates a broad range of type management styles in multi-environment systems (*cf* section 3.5.5). The interrupt condition, with priority-based synchronization, is more transparent than either Concurrent Pascal or Modula, and still provides mutual exclusion. All three case studies dealt with real-world hardware and software systems. We were able to develop a hierarchical description of the VAX/VMS process and memory management facilities without changing their implementation. The bootstrapping description, likewise, was faithful to the actual VAX/VMS design. Although some system designs are not worth saving, if the methodology applies to designs that were developed without its benefit, then one has some confidence that the methodology does not constrain the design space unduly.

7.4. Implementability

We have deferred until this time almost all discussion of implementation issues. Although a detailed implementation of the language support tools implied by the methodology would be premature, we seek some indication that they are feasible and practical. To investigate this, we shall sketch a design for the program support facility, then consider each of the implementation problems implied by the overall design.

An operating system written under the proposed methodology could conceivably be written as a single program, that is translated and elaborated as a single unit by an appropriate compiler. However, the size of the system dictates that we be able to decompose it into separately compilable modules for purposes of development and testing, and that we be able to recompile selected portions of the system and reintegrate them with previously compiled components. Walter Tichy [Tichy 80] has already shown how to control the development of large software systems through a module interconnection language, including automating selective recompilation. Special problems posed by multi-environment systems include:

- The system integration phase is explicitly programmed rather than following the standard compile-link-load sequence. How can the language system integrate incremental changes without completely retranslating the system?

- How much code optimization can be carried out between separately compiled modules?

- How shall the translator accommodate variations in the implementation language supported in different environments? How shall the translator make effective use of user-supplied environment implementation software?

7.4.1. Separate Compilation

Compilation under the proposed environment management paradigm is different than the *status quo* in the following ways:

- Translating a type definition *never* implies any code generation. Only *instantiating* a type or translating a *module* does.

- The size and contents of a permanent data structure may fluctuate during translation of subsequent system components, prior to linking the data structure into a segment.

- Consecutive program units within a single file may reside in different segments

To accommodate these differences, we identify the following stages in the translation of a program unit:

1. Specification processing: extracting the syntactic information needed to check the syntactic correctness of other units.

2. Parsing: translating the program text into an internal representation (presumably syntax trees, a symbol table, and some representation for permanent variables), and verifying its consistency with the specifications of other units.

3. Code generation: producing "object code" representation of individual procedures (leaving unresolved external identifiers), and constructing an interpretable representation for the sequence of actions implied by elaborating the program text.

4. Initialization: carrying out the actions implied by elaborating the program text, including storage allocation and explicit initialization code.

5. Linking: finalizing the layout of permanent data structures, determining the logical addresses of code and data objects, and resolving external references. This phase is radically different from the previous ones, in that linking happens simultaneously to all objects in a *segment*, whereas each previous phase happens at the same time to all objects in a *module*. Linking, in fact, is simply an abstract operation on a *segment*, carried out by host-environment procedures of the module responsible for the segment, usually as part of the down-loading phase initiated by the the outermost module.

With this model of translation in mind, we may determine which phases must be

carried out simultaneously for the whole system, which phases may be carried out sequentially, and which phases can be carried out for each module independently.

1. Specification processing: carried out independently for each program unit. Programming languages like Ada allow the specifications to be compiled before the implementation is even written.

2. Parsing: a program unit may be parsed as soon as all specifications have been collected for the program units it invokes. Changing the implementation of a program unit without changing its specification will not force reparsing of any other program unit.

3. Code generation: code for a program unit may be generated as soon as parsing is complete for the program units it uses. Generated code is not placed in segments until the initialization stage.

4. Initialization: the program units must be initialized in the order in which they appear in the system description. Forward references during initialization might not be feasible (an open research/engineering question). On the other hand, there may be large classes of initialization actions that can be carried out covertly during code generation.

5. Linking: linking a segment terminates host-environment access to the variables contained in it. A segment can be linked any time this is acceptable; the actual time of linking must be programmed explicitly as part of the host machine activities of the operating system. Presumably there will be portions of the linking activity that can be carried out earlier, and will not need to be redone after each incremental change.

Thus we see that each of the first three stages can be carried out on each module independently, as soon as the modules upon which it depends have finished the previous phase. Only during the initialization must the modules be processed sequentially. Even then, the state of the system may be saved at any point, so that the state of the earlier (lower-level) system components need not be reinitialized after a change to a later part. In particular, the segments containing code and data for lower system levels might be linked quite early in the initialization phase, and therefore rarely need to be relinked.

7.4.2. Inter-module optimization

Under the translation paradigm given above, the code generator has access to the syntax-tree representation of all of the program units being invoked. Depending on the sophistication of the code generator, this allows inline expansion of procedures, optimization of access functions, and so on. Some value-dependent optimizations may be precluded because the values are not known until initialization. Inter-environment optimizations are usually possible, except when a procedure or data structure is explicitly bound to a particular environment.

7.4.3. Customized Language Run-time Support Software

We have hypothesized the ability to use different implementation languages in different execution environments within our system. This will be practical only if the languages are closely related dialects of a single language. Variations should only be necessary to remove unsupported features and accommodate peculiarities of the operating system design itself. The case studies presented in the last three chapters have so far permitted the use of a single language throughout, except that some features might not be available in some environments. Should the need arise to specify a dialect, it could be defined in the environment module interface. For example, whether or not a particular feature is available to a particular program unit depends primarily on whether or not the support module is present in the intended execution environment. Thus a single compiler could implement many subsets of a language, where the subset is defined by the *provides* clause of the environment module.

The translator applies support software for a given language feature by translating uses of the feature into invocations of the support module. First it parses the user and supporter of a given f ature separately. The user must adhere to the language syntax of the feature; the supporter must adhere to the language specification for the support module. Then, the parser replaces each occurrence of the feature in the parse tree with the appropriate invocation of its implementation. (Recall that support modules do not export any names that are reserved to the translator, so that a program may use these names only via the language feature. See, for example, Sten Andler's implementation of abstract types in Algol68S [Andler 79].) The parser having established the connection, the code generator can treat the invocations just like ordinary procedure calls and data accesses.

Undoubtedly there are a great many implementation difficulties that will not surface until an implementation is actually attempted. Nonetheless, by dividing the translation into the stages identified above, the program development environment can support incremental recompilation, inter-module optimization, and program-supplied language support software.

7.5. Summary

For each of the criteria established in Chapter 1, we have found the methodology satisfactory:

- It is *useful* for solving both theoretical and practiacl problems.
- It *clarifies* system descriptions, with notations that *fit* their problem domains.
- It is *flexible* enough to support a broad range of design approaches, facilitating system construction without constraining system design.

- It appears to be *implementable* with available techniques, albeit applied in novel combinations.

CHAPTER 8
CONCLUSIONS

This thesis presents a methodology for describing the executable representation of a program, and uses the methodology to investigate significant description problems in operating systems. The methodology integrates an operating system with its implementation language, so that the language can facilitate use of system components without interfering in their design. The system designer can define execution environments as *bona fide* source program entities; each is a set of resources that the language system uses to implement programs. The methodology improves the system designer's ability to describe many system properties directly in the program text, as demonstrated in the areas of interrupt synchronization, bootstrapping, and hierarchical structure.

To conclude this thesis we shall first compare the methodology to other work in operating system design methodology, then summarize the contributions of the work, and discuss future directions the research might take.

8.1. Relationship to Other Work

References to other work are scattered throughout this thesis, providing a basis for developing new techniques. In this section we shall compare the outcome of the thesis to the ideas and techniques previously known. The discussion is organized by topic rather than by project. Some projects appear under more than one topic.

8.1.1. Implementation Languages Supporting Synchronization

Concurrent Pascal, Modula, and Gypsy all provide synchronization mechanisms explicitly in the language syntax. Concurrent Pascal provides *monitors*, Modula provides *interface modules*, and Gypsy provides *mailboxes*. Each of these mechanisms requires runtime support, supplied by a language support kernel defined outside the language.

Such language support kernels are actually supplying a portion of the operating system, namely an execution environment that supports cooperating sequential

processes. Under the proposed methodology the language definition could specify the facilities the language kernel would supply, but leave the implementation of the kernel to the operating system designer.

These languages incorporate device communication into the same synchronization mechanism. Concurrent Pascal represents its I/O devices as hypothetical processes communicating with the program via queues and signals. Modula defines a *device interface module* which contains a *device process*. That process may communicate only with ordinary processes. The interface module enforces mutual exclusion using priority. An interrupt process cannot cause pre-emption.

The interrupt condition module defined in chapter 4 is more transparent more flexible, and more fit for device communication than the mechanisms described above, while still providing an acceptable synchronization mechanism. An interrupt is modeled by a procedure call rather than a signal or message, allowing the interrupt routine to pre-process the arriving information without the overhead of a full process context switch. Two devices may communicate directly via shared variables, again without the overhead of context swapping. Traps and programmed interrupts can be described with the same notation, which is appropriate considering that they are defined analogously in most architectures.

Peter Loehr, describing his attempts to write a virtual memory operating system in Concurrent Pascal [Loehr 77], claims that systems implementation languages should have fewer specific mechanisms and greater extensibility. Although I agree with his reactions to Concurrent Pascal, a context-sensitive language notation can facilitate static analysis and enforce methodological principles in ways that run-time mechanisms cannot. The methodology allows the system designer to select the language mechanisms appropriate to the system being designed, and to retain control over the implementation of the mechanisms.

8.1.2. Hardware Access in High-Level Languages

Euclid and Modula both give programs access to hardware-defined device registers, by means of special variable declarations. I presume such a feature in my implementation language, and use environment definitions to specify which program units may declare such variables or access them. Rather than have each individual device register declared within the module that manages it, I declare all hardware-defined objects in modules corresponding to their hardware implementations, then let the managers *acquire* the objects they control.

8.1.3. Process and Memory Management Methods

Janson observed that environment management greatly complicated the structure of Multics, threatening to make the dependency graph cyclic. For example, he thought he might need capability lists to implement capability lists. He resolved this problem by defining the notion of a "map" dependency, and then differentiating between a/f types (allocated and freed as needed) and c/d types (created and deleted at will). He viewed a/f types as more fundamental than c/d types. Reed [ReedThesis] proposed a similar scheme for process management

In my hierarchy case study I defined the scheduler in such a way that it would work correctly regardless of how many process descriptors existed, and regardless of whether the number was fixed or varying. The swapper determined that there would be a fixed number of them, but could easily have made them dynamically allocated and deleted. Thus the multiplexor did not need to be responsible for the size of the management set.

Incorporating the host environment in the operating system description explains the relationship between permanent and non-permanent instances of a type, allowing them to be derived from the same type definition but instantiated from different resource pools.

8.1.4. "User-supplied" Runtime Support

Euclid storage zones illustrate how a language can specify the characteristics of its runtime support system, letting programs supply their own implementations of the support. My methodology generalizes this technique to cover all of the basic environment features, plus the *condition module*.

8.1.5. Module Interconnection Languages

Tichy's module interconnection language and processor, and the Gandalf system, provide the basis for the software development control system needed for operating systems. It allows control over the development of multiple serial and parallel versions, keeping enough information to minimize the amount of retranslation needed to reconstruct a runnable system after a program change. However, Tichy's scheme presumes a classical compile-link-load scheme. My comprehensive system description replaces that scheme with a more detailed breakdown of the translation steps: process specifications, parse, generate code, and initiate (including linking and loading). The modified translation scheme does not materially impact the other Gandalf facilities for software development control, allowing the operating system designer to obtain the same benefits available to single-environment systems.

8.2. Contributions of the Thesis

This thesis contributes three distinct kinds of knowledge about operating system design and implementation:

- Ideas about the relationships between langauges, execution environments, and operating systems.
- Methods for describing the executable representation of programs,
- Solutions to three diverse operating system description problems,

8.2.1. Ideas about Execution Environments

A systems implementation langauge can facilitate system construction by providing notations for multi-tasking, synchronization, device communication, and so on. However, in doing so, the language imposes constraints on the operating system design. Therefore, the system designer must choose (or design) an impelementation language that harmonizes with the design he envisions. The operating system then becomes the "run-time system" for the language. Execution environments define the interface between the language and the operating system.

An execution environment is a complete set of operating system resources that together provide everything the language needs to implement programs. We distinguish the concept "execution environment" both from the concept "surrounding scope" and the concept "runtime protection domain". Every module has a different "surrounding scope", whereas an operating system contains a relatively modest number of coherently-designed execution environments. "Runtime protection domains" define the possible actions of arbitrary machine language programs, whereas operating systems programs are (or ought to be) written in strongly typed languages with powerful tools for defining and enforcing modularity. By defining execution environments explicitly in source-language terms, the system designer gains the ability to write multi-environment modules and multi-module environments, and to choose a blend of compile-time and run-time protection mechanisms for enforcing the boundaries of execution environments.

8.2.2. Methods for Describing Executable Representations

The exact details of the proposed methods are highly speculative, and I would expect them to change considerably when tested experimentally. Nevertheless, I expect the following principles to endure:

- An execution environment appears in a source program as an explicit list of facilities, such as a module.
- That list defines the interface between the compiler and the operating system, for all program units residing in that environment.

- A translator is a type manager for the abstract type "program unit", implementing it with the resources supplied by the operating system. The particular environment supplying the resources for a given program unit is a generic parameter to that instance of the type.
- The host environment and bootstrapping environments are *bona fide* components of the operating system.
- A complete system description is a host-machine program to create an operating system.

8.2.3. Specific Solutions

The interrupt condition mechanism of Chapter 4 is a strongly typed yet highly transparent characterization of interrupt hardware. It provides an acceptably powerful synchronization tool without pre-empting operating system design decisions.

In designing this mechanism we studied the tension between abstraction gained and transparency lost when a language provides a synchronization mechanism. We observed that a context-sensitive notation can facilitate static analysis and enforce methodologically sound design priniciples in ways that the runtime support for the mechanism cannot. From this we conclude that systems implementation languages *should* include such features, even though they constrain system design; choosing the language features should be part of the operating system design process.

In chapter 5 we outlined the design of the VAX/VMS bootstrapping mechanism. We were able to embed it in the same module structure that defines the running system. We used the *startup condition* to pass control from lower to higher system levels without violating hierarchy. We used the host environment to create permanent and paged segments using the same type definitions as are used in the running system. The startup condition provided a place to represent the memory map enabling trick and the mechanism for disposing of startup code. Overall, we found that every piece of the bootstrapping mechanism of VAX/VMS had a place in the system description.

We were only able to look briefly at the problem of defining handler initialization and finalization protocols. Further investigation in this area will benefit not only bootstrapping, but also exceptional condition handling generally.

In chapter 6 we developed a demonstrably hierarchical description of a real operating system. By using source language type checking and modularity for protection, we allowed two execution environments to share the system page table. This broke the cycle in the dependency graph between process multiplexing and demand paging.

To resolve the distinction between process multiplexing and process creation and

deletion, we defined a generic scheduling module that was insensitive to the number and location of process descriptors. *Without* environment bindings, we would have informally placed the burden of addressability on the module instantiating a particular scheduler. *With* environment bindings, we could specify directly the addressing constraints for the scheduler code, process descriptors and process state vectors.

The *acquires* clause defined in section 3.2 supplies a source-language replacement for a runtime protection paradigm. In the Family of Operating Systems we introduced multi-level modules as a way of explaining that certain facilities provided by the lower level were accessible to only those upper level procedures residing in the same module. By using the *acquires* clause to declare exclusive access, we can define protection environments independently from source language modules, using the visibility of the environment name to specify which modules may place code and data in it.

More generally, the *acquires* clause expresses more precisely the *composition* relation in a system description. Statically nested program units by themselves support only those composition relations that are trees. The *acquires* clause supports composition relations that are directed acyclic graphs.

8.3. Future Directions

Demonstrating that explicit execution environments clarify the structure of operating systems opens up many avenues for future research, including

- Refining the concept of an execution environment
- Implementing the translation paradigm, especially the initialization phase executing in the host environment.
- Designing and implementing a systems implementation language that includes notations for environment bindings, exceptional conditions (including interrupt and startup conditions), capabilities, synchronization, multiple processors, protected procedures, and other operating system facilities.
- Operating systems implementation experiments to test the usefulness of the methodology.

References

[Ambler 77] Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells.
Gypsy: A Language for Specification and Implementation of Verifiable Programs.
In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 1-10. Association for Computing Machinery, Inc., March, 1977.

[Andler 79] Andler, Sten A.
Predicate Path Expressions: A High-Level Synchronization Mechanism.
PhD thesis, Carnegie-Mellon University, 1979.

[Brinch Hansen 75] Per Brinch Hansen.
The Programming Language Concurrent Pascal.
IEEE Transactions on Software Engineering 1(2):199-205, June, 1975.

[Cattell 78] R.G.G. Cattell.
Formalization and Automatic Derivation of Code Generators.
PhD thesis, Carnegie-Mellon University, April, 1978.

[DeMillo 79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis.
Social Processes and Proofs of Theorems and Programs.
Communications of the ACM 22(5), May, 1979.

[DeRemer 75] Frank DeRemer et al.
Programming-in-the-Large vs. Programming-in-the-Small.
In *Second International Conference on Reliable Software.* ACM Special Interest Group on Programming Languages, 1975.

[Dijkstra 68] Edsger Dijkstra.
The Structure of the 'THE' --Multiprogramming System.
Communications of the ACM 11(5), May, 1968.

[Dijkstra 76] Edsger Dijkstra.
A Discipline of Programming.
Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Flon 77] Lawrence Flon.
On the Design and Verification of Operating Systems.
PhD thesis, Carnegie-Mellon University, 1977.

[Goullon 78] Goullon, H., R. Isle and K.-P. Loehr.
Dynamic Restructuring in an Experimental Operating System.
IEEE Transactions on Software Engineering 4(4):298-307, July, 1978.

[Habermann 76] Nico Habermann, Lawrence Flon, Lee Cooprider.
Modularization and Hierarchy in a Family of Operating Systems.
Communications of the ACM 19(5), May, 1976.

[Habermann 78] Nico Habermann et al.
Modularization and Hierarchy in a Family of Operating Systems.
Technical Report CMU-CS-78-101, Carnegie-Mellon University, Computer Science Department, February, 1978.
Final report of the project.

[Hoare 72] C. A. R. Hoare.
Proof of Correctness of Data Representations.
Acta Informatica 1(4):271-281, 1972.

[Janson 76] Philipe Janson.
Using Type Extension to Organize Virtual Memory Mechanisms.
PhD thesis, Massachusetts Institute of Technology, August, 1976.
MIT LCS TR 167.

[Levin 77] Roy Levin.
Program Structures for Exceptional Condition Handling.
PhD thesis, Carnegie-Mellon University, June, 1977.

[Loehr 77] Klaus-Peter Loehr.
Beyond Concurrent Pascal.
In *Proceedings of the Sixth Symposium on Operating Systems Principles*, pages 173-180. Special Interest Group on Operating Systems, November, 1977.

[Needham 77] R.M. Needham, R.D.H. Walker.
The Cambridge Cap Computer and its Protection System.
In *Proceedings of the Sixth Symposium on Operating Systems Principles*, pages 1-10. Special Interest Group on Operating Systems, Purdue University, West Lafayette, In, November, 1977.

[Parnas 71] D.L. Parnas.
On the Criteria to be used in Decomposing Systems into Modules.
Technical Report CMU-CS-71-101, Carnegie-Mellon University, Computer Science Department, August, 1971.

[Parnas 72a] David L. Parnas and D. Siewiorek.
The Use of the Concept of Transparency in the Design of Hierarchical Systems.
Technical Report, Carnegie-Mellon University, Computer Science Department, September, 1972.

[Parnas 72b] David L. Parnas.
Information Distribution Aspects of Design Methodology.
Technical Report, Carnegie-Mellon University, Computer Science Department, 1972.

[Parnas 74] David L. Parnas.
On a Buzzword: 'Hierarchical Structure'.
Technical Report, Carnegie-Mellon University, Computer Science Department, 1974.

[Parnas 78a] David L. Parnas and John E. Shore.
Language Facilities for Supporting The Use Of Data Abstractions In The Development of Software Systems.
Technical Report, Naval Research Laboratory, February, 1978.

[Parnas 78b] David L. Parnas.
The Non-problem of Nested Monitor Calls.
Operating Systems Review 12(1), January, 1978.

[Reed 76] D. Reed.
Process Multiplexing in a Layered Operating System.
Master's thesis, Massachusetts Institute of Technology, June, 1976.
MIT LCS TR-164.

[RSX 78]
RSX-11-M Overlay Description Language.
Reference Manual.

[Schwanke 78] Robert W. Schwanke.
Survey of Scope Issues in Programming Languages.
Technical Report CMU-CS-78-131, Carnegie-Mellon University, Computer Science Department, June, 1978.

[Tichy 80] Tichy, Walter F.
Software Development Control Based On System Structure Description.
PhD thesis, Carnegie-Mellon University, 1980.

[Wirth 80] Niklaus Wirth.
Modula-2.
Technical Report, Eidgenossische Technische Hochschule, March, 1980.
ETH technical report.

[Wulf 74a] Wm A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack.
Hydra: The Kernel of A Multiprocessor Operating System.
Communications of the ACM 17(6):337-345, June, 1974.

[Wulf 74b] Wm A. Wulf.
Private Communication to David L. Parnas.

[Wulf 76] W.A. Wulf, R.L. London, and M. Shaw.
Abstraction and Verification in Alphard: Introduction to Language and Methodology.
Technical Report, Carnegie-Mellon University, Computer Science Department, June, 1976.