Linkage in the Nemesis Single Address Space Operating System

Timothy Roscoe
Computer Laboratory,
University of Cambridge

May 9, 1994

Abstract

The recent interest in single address space operating systems has resulted in a number of papers, most of which gloss over the issues of linking programs to run in multiple protection domains. Some of the confusion about 64-bit address spaces is due to the almost pervasive use of UNIX and UNIX-like operating systems (such as Mach, Chorus and Amoeba) and languages with poor enforcement of abstraction like C and C++.

This paper describes some of the linkage structure of Nemesis, a multi-service operating system being developed as part of the Pegasus project. Nemesis provides a simple and efficient mechanism for program linkage which provides rich sharing of text at a level of individual object classes.

1 Motivation and Aims

Nemesis is a multi-service operating system being developed as part of the Pegasus ESPRIT Project ([**Leslie93**]). Nemesis is designed to allow sharing of data and text with as little overhead as possible for the efficient processing of continuous media data, while reducing quality-of-service crosstalk between application domains by moving as much functionality as possible into the application domains. This is in contrast to microkernel approaches where the functionality has been moved into server processes [**Roscoe94**]. This coupled with the use of DEC Alpha machines as one of the target architectures led to the idea of using one address space throughout the operating system.

This paper does not attempt to describe novel features of Nemesis in the area of resource management. Instead it examines the method of linking and loading programs and system components which we use to support the rest of the system. The fundamental concepts in single address space operating systems are not new: OS/360 ([**Witt66**]) and Cedar ([**Swinehart86**]) are two very different examples from some time ago. Similarly there are almost no issues in naming objects within a single operating system which are not addressed fairly comprehensively in [**Saltzer79**]. Within the Pegasus Project the aim is to build an operating system to support a Quality-of-Service paradigm for resource allocation and per-application resource management. With this in mind, we are applying the single address space idea as an enabling technology for handling a variety of different multimedia-related application types efficiently. We do *not* believe 64-bit address spaces are a panacea, and our motivation for building such a system differs from many other researchers in the field (for example [**Wilkinson92**], [**Heiser93**] and [**Assenmacher94**]). In particular:

- The address space is not distributed. Distributed shared virtual memory does not scale and obscures too many performance issues compared with Remote Procedure Call. RPC with a suitable programming model such as that described in [**Birrell93**] offers better abstraction and makes remote operations

explicit when necessary. Coherent distributed memory, if required, can in any case be implemented in user space on a per-segment basis (the Nemesis virtual memory system runs largely as a shared library).

- Persistent address-space objects are not an aim of the project. Again, user-level support for persistence can be provided in Nemesis as part of an application's virtual memory system.
- We are not interested in providing backward compatibility with operating systems with different paradigms. This can be done properly with a shared library and some compiler technology, and we do not consider it worthwhile to bend our operating system architecture unnecessarily. Interestingly, UNIX emulation turns out to be quite easy in our system despite it not having been a design requirement.

2 Single Address Space Issues

The recent resurgence of interest in single address space systems has caused people to rediscover many naming issues which have been obscured by the almost ubiquitous use of C, C++ and UNIX-like operating systems.

In particular, being able to assume that a program has the whole address space to itself means that in C or C++ the execution environment for any procedure call is generally the sum total of data in the program. Aside from encouraging poor programming abstraction, this is disastrous in a single address space.

The naïve solution to this problem is to simulate per-process address spaces by keeping around a pointer (sometimes in a dedicated processor register) to a per-process data segment which contains most of the program state. Apart from being a throwback to the earliest days of multiprogramming, this has a number of disadvantages:

- The granularity of code sharing is very coarse: all the code constituting a program must be aware of the structure of the data segment. This means in practice that either you can only share text at the level of entire program binaries (as in UNIX), or every component of loaded code must assume the same format of data segment, which is absurd.
- To regain one of the main advantages of a single address space, namely that one can pass pointers between processes, there have to be two classes of pointers: both conventional absolute addresses and offsets from the start of the data segment ([**Wilkinson93**]). This is unacceptable.

Clearly a better approach is required. The problem is really only one of explicitly specifying the calling environment to a procedure.

3 Interfaces

At an early stage in the Pegasus project it was decided to define all interface types[1] in the system in an interface definition language which came to be known as *Middl*. While aiding system development this has another benefit: it naturally leads to a model of programming where an invocation across an interface passes a reference to the interface as an argument. The interface structure in memory contains a reference to state unavailable to the client. This state together with the other arguments to the call constitute the complete calling environment. This device is known as a closure. The style of programming is known as object-based.

A closure is a pair of pointers; one points to some state record, the other to a method table within a module (see figure 1). Since interfaces are typed, the format of the method table can be derived from the

[1] In Nemesis, an *interface* is an instance of a particular *interface type*.

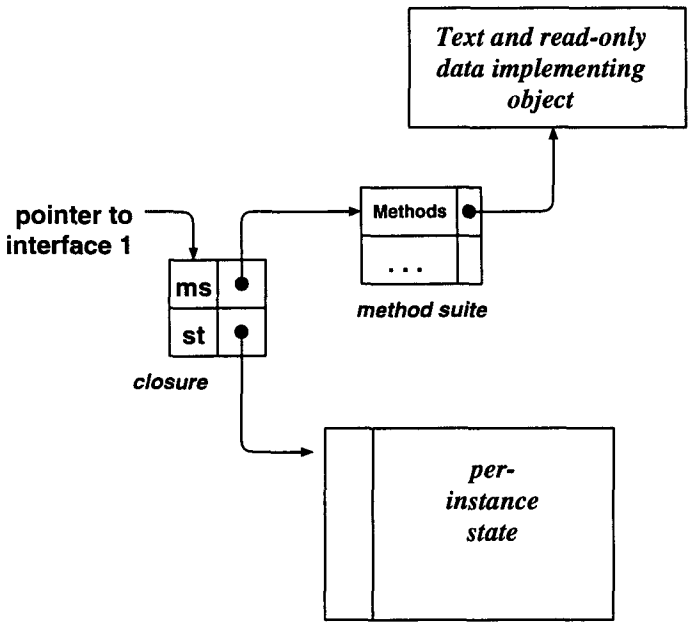

Figure 1: Interface data structures

interface type specification. The state record is opaque to any client of the interface; it is only manipulated by code within the module. Any call across an interface adheres to a particular calling convention, and must pass the closure as the first argument. Thus the complete environment for the call is passed as arguments.

This technique solves the problem of identifying the calling environment: it is the state in registers and that accessible on the call stack or via pointers contained therein. All mutable state in a domain is encapsulated behind interfaces instantiated at runtime. In some ways our system is similar to Opal ([**Chase92**]), although the use of Middl allows a rather more flexible computational model (for example, we can have multiple interfaces attached to object state, efficient cross-module exception handling, and a cleaner syntax for multiple return values). Nemesis is also a native operating system, rather than one running over another operating system such as Mach.

Writing programs in this manner is not terribly alien; closures are after all the mechanism used by many object-oriented languages. As a result, the performance penalty due to the indirection involved is no more than virtual function lookup in C++, for example.

By making closures explicit we gain modularity in the system and at the same time do away with the many of the problems of addressing state in the system. We can share code at a fine level of granularity and freely pass real pointers between components of the system with compile-time type checking.

At a pinch, we can support existing UNIX-oriented C and C++ programs in the same way that other operating systems do: by linking the program with a set of stub functions with interfaces to our libraries and running the whole thing in a single call environment.

4 Linking and Loading

Like many systems, Nemesis separates the notions of linking and loading: linking is the process of resolving internal references within a chunk of object code, while loading is the business of relocating it and installing it in the address space of a Nemesis system. The model of linking and loading adopted in Nemesis needed

to:

- solve the addressing issues inherent in a single address space operating system.
- deliver high performance at runtime, and
- allow fine-grained sharing of program text between protection domains.

Our solution is centred around the concept of modules. A Nemesis module is a self-contained chunk of code. It has no unresolved symbols or mutable data, and when loaded is potentially executable by any domain. All code in Nemesis (with the exception of the minimal assembler kernel) is part of some module.

The resulting programming model is one of "objects"[2] which export multiple interfaces, each of which is identified by an address. Every call across an interface (in effect, any call across a module boundary) must pass the interface address as a closure. Objects *per se* don't have types; their interfaces do. In this way the notions of type and class are separated: the type of an interface is the set of operations it supports, and the class of an object is its implementation, which depends on the module that implements it. One way to think about modules is as CLU clusters ([**Liskov81**]), except that modules can have no "own" variables since they are shared read-only between all domains.

We have found programming in this way is actually quite natural in stylised C or C++, especially with Middl compiler tools to generate the boilerplate. There is also design work under way to produce a language which compiles naturally down to C, but which maps very closely onto our computational model and provides more type safety and compile-time protection than C++ or C.

5 Runtime issues

Because we expect to support a wide variety of different policies for scheduling, paging, etc. the idea of a common, ubiquitous runtime is not applicable in the operating system. Thus there is a problem of how to create objects and interfaces at the start of day—for example we can't do the equivalent of C++ `new` since there is no notion of where the heap state is.

Interfaces which are inherently stateless present no problem: a closure with a null state pointer can already be executed in any domain. Thus further objects can be created by explicitly passing in closures which implement the functions normally provided by a runtime system, with greater control over the resource tradeoffs used (such as designating a heap to allocate storage from). A typical object constructor takes a set of interfaces as arguments (possibly with some additional parameters) and returns all the interfaces exported by the new object. The idiom has proved so useful that we are considering giving it language-level syntax. Most modules are built with only one externally visible symbol, namely the closure for a constructing objects of the class that the module implements (see figure 2). The address of this interface is registered with a Nemesis name service when the module is loaded into the system.

Starting up an application domain requires some initial state to be setup, though in practice surprisingly little is required. Domains are started by a Nemesis service called the Domain Manager, which instantiates an initial set of interfaces. These include a memory protection domain to run the application in, a basic memory allocation heap and communication channels to a name server and the inter-domain communication binder.

These interfaces encapsulate all the mutable state required by an application when it starts up, and are passed as parameters to the domain entry point. This is a stateless closure within a module which then instantiates the rest of a domain's paraphernalia (scheduler, language run-time, application-specific state, etc.).

[2] for want of a better word.

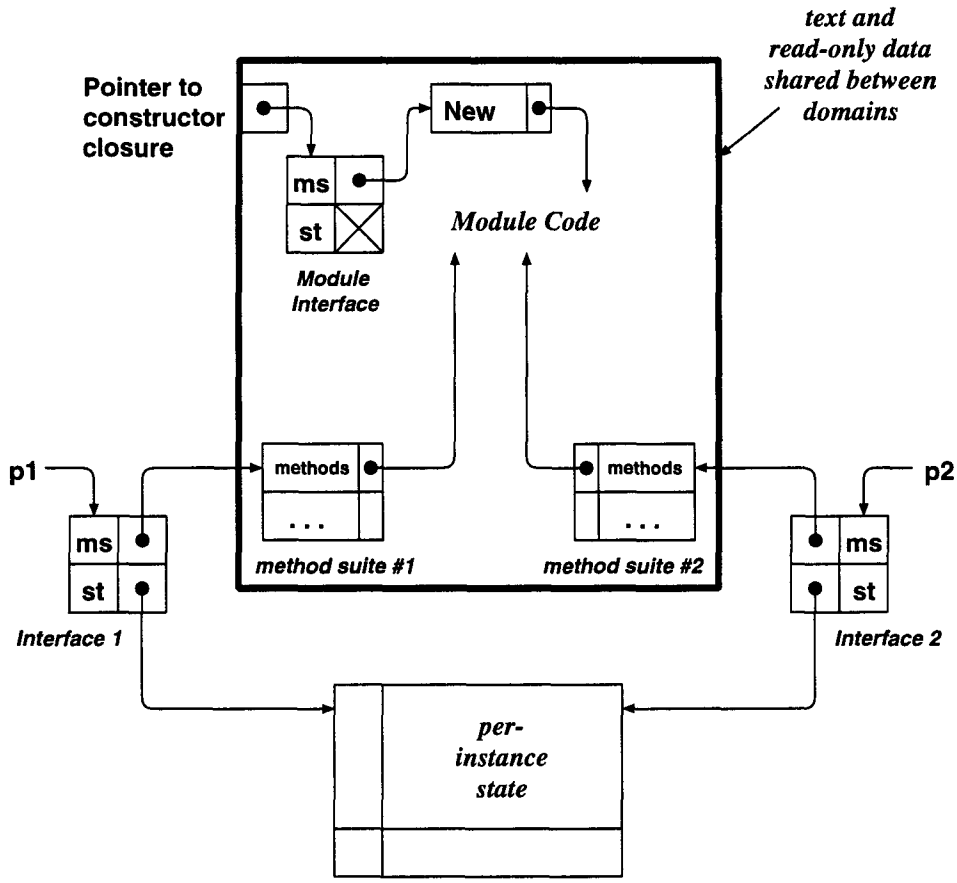

Figure 2: A module implementing an object with two interfaces

6 A Concrete Example

Consider a body of code to implement a hash table. A programmer's interface to the table might be specified in Middl as follows[3]:

```
HashTable : INTERFACE =
BEGIN
  Enter : PROC [ n : INTEGER, key : STRING ]
    RETURNS [ ];

      [...]

END.
```

The Middl compiler would generate C defining the following data structures:

```
struct HashTable_op {
  void (*Enter)(struct HashTable_cl *self, int32 n, char *key);
```

[3]This is a considerably simplified specification; in particular many features of a real interface type (such as supertypes, exceptions, etc) have been omitted for clarity.

```
  ..
};

struct HashTable_cl {
  struct HashTable_op *ms; /* ptr to method suite */
  void                *st; /* ptr to state        */
};
```

An instance of the hash table would be identified by a pointer to a `HashTable_cl` structure. This structure's `ms` pointer would point to a constant `HashTable_ms` structure, whose fields in turn would point to the code implementing the table. A method invocation in C might would look like:

```
ht->ms->Enter(ht, 10, "Token");
```

—where `ht` is a pointer to a Hash Table closure[4].

The code implementing the hash table might reside in a module called **HTMod**. This would include the code for the methods, the method suite struct, and a further method suite and closure (with null state pointer) for the following interface:

```
HTMod : INTERFACE =
  NEEDS HashTable;
  NEEDS Heap;
BEGIN
  New : PROC [ h : IREF Heap ]
    RETURNS [ ht : IREF HashTable ];
END.
```

(To create a Hash Table we need to supply a closure for a Heap object, and we are returned a closure to the newly created hash table).

The only externally visible symbol in the module would be the **HTMod** closure. The address of this struct is registered with the operating system's name server when the module is loaded. The code for the **New** method of the **HTMod** interface is completely stateless, thus it can be executed in any domain.

7 Pervasive Interfaces

It becomes very clear that certain interfaces within an application are used in almost all parts of the program. These include to some extent language support functions (such as those found in the C, C++ or Modula-3 runtime libraries). More important, however, are those interfaces which are inherently pervasive: these include the current thread, the user-level scheduler, and the minimal kernel. These are still notionally passed as call arguments, though in practice their use makes them seem more like part of the application's top-level context. It seems sensible to produce a convention (or a number of different conventions) whereby a well-defined set of interface references are passed implicitly with every procedure call. The implementation of this can then by optimised, for example by keeping the address of a record of these interfaces in a processor register. It is important to note the differences between this optimisation and the data segment pointer approach to text sharing described in section 2.

Firstly, only a small set of interface references are being kept. The use of these interfaces is so ubiquitous that they would be passed as arguments to every call in any case, or else copies of them would be maintained in many interface state records. In contrast, the data segment register approach keeps pretty much all the program state in one place, in a format which is highly application specific.

[4] In practice syntactic sugar would be laid over this by a preprocessor tool.

Secondly, what are being kept are still interfaces: their types are defined in Middl and the complete set of them available is "written on the wall" in the system for all to see. They represent an abstraction boundary, rather than being raw data.

Thirdly, they are there for convenience only. The basic philosophy remains the same and modules may choose to ignore the presence of pervasives. In particular, the low levels of the operating system do not use them at all and pass all interfaces explicitly.

One useful side effect of this is that standard components of a program can be replaced at runtime. It is even possible to instantiate a set of modules implementing a program in an entirely "caged" environment, where even the operating system is being emulated, while other instantiations of the same program sharing the same text run "native". This has obvious uses for debugging purposes, and is a considerably cleaner approach than linking with debugging libraries.

8 Higher-level Naming

Machine addresses provide unique identifiers for interfaces within one machine, but a higher-level naming scheme is required for several reasons, for instance:

- We want to refer to entities in distributed systems, which have rather different naming requirements.
- On a single machine, programs do not know addresses of other components when they are linked. These must be determined using other names at load time and run time.
- Human-readable names are required for users, managers configuring the system or examining it remotely, and programmers building and debugging it.

The important thing to realise is that these naming issues are *completely orthogonal* to the idea of a single address space, and so almost any existing name space scheme can be used. Our approach is currently based on a directed graph of contexts referred to with pathnames, together with a simplified version of the ANSA Trader's constraint language ([**ANSA92**]). This allows us a lot of expressive power in the name space when we need it (and extends naturally to the distributed case), but in well-known contexts with few or no constraints name lookup can be extremely fast.

9 Conclusion

To take full advantage of a single address space paradigm when building an operating system does not need any radically new ideas, but does require that one steps back from the UNIX/C mentality and considers more carefully what is really going on.

We have presented a system which provides efficient, rich sharing of text and data within a machine and provides a nice computational model which is quite natural to programmers used to an object based style. It provides type safety and flexible dynamic loading. We are using this as a means of constructing an operating system designed to support the resource demands of distributed multimedia applications.

References

[**ANSA92**] Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK. *ANSAware 4.0 Application Programmer's Guide*, March 1992. Document RM.102.00. (p 7)

[**Assenmacher94**] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. *The PANDA System Architecture—A Pico-Kernel Approach.* Technical Report, University of Kaiserslautern Department of Computer Science, P.O. box 3049, 67653 Kaiserslautern, Germany, 1994. Available via ftp from drei.informatik.uni-kl.de:/pub/panda. (p 1)

[**Bayer79**] R. Bayer, R. M. Graham, and G. Seegmuller, editors. *Operating Systems: an Advanced Course*, volume 60 of *LNCS*. Springer-Verlag, 1979. (p 8)

[**Birrell93**] Andrew Birrell, Greg Nelson, Susan Owicki, and Ted Wobber. *Network Objects.* Proceedings of the 14th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review, 27(5):217–230, December 1993. (p 1)

[**Chase92**] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. *Lightweight Shared Objects in a 64-Bit Operating System.* In Proceedings of 7th OOPSLA Conference, volume 27 of *ACM SIGPLAN Notices*, pages 397–413, October 1992. (p 3)

[**Heiser93**] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. *Mungi: A Distributed Single Address-Space Operating System.* Technical Report 9314, School of Computer Science and Engineering, The University of New South Wales, November 1993. (p 1)

[**Leslie93**] I. M. Leslie, D. R. McAuley, and S. J. Mullender. *Pegasus — Operating System Support for Distributed Multimedia Systems.* ACM Operating Systems Review, 27(1):69–78, January 1993. (p 1)

[**Liskov81**] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *LNCS*. Springer-Verlag, 1981. (p 4)

[**Roscoe94**] Timothy Roscoe. *The Structure of a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, 1994. In preparation. (p 1)

[**Saltzer79**] J. H. Saltzer. *Naming and Binding of Objects.* In Bayer et al. [**Bayer79**], chapter 3.A, pages 100–208. (p 1)

[**Swinehart86**] D. Swinehart, P. Zellweger, R. Beach, and R. Hagemann. *A Structural View of the Cedar Programming Environment.* Technical Report CSL-86-1, Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304, June 1986. (p 1)

[**Wilkinson92**] T. Wilkinson, T. Stiemerling, P. Osmon, A Saulsbury, and P. Kelly. *Angel: A proposed multiprocessor operating system kernel.* Technical Report TCU/CS/1992/10, Department of Computer Science, City University, London, 1992. (p 1)

[**Wilkinson93**] Tim Wilkinson, Ashley Saulsbury, Tom Stiemerling, and Kevin Murray. *Compiling for a 64-bit Single Address Space Architecture.* Technical Report TCU/SARC/1993/1, Systems Architecture Research Centre, City University, London, March 1993. (p 2)

[**Witt66**] B. I. Witt. *The Functional Structure of OS/360 Part II: Job and Task Management.* IBM Systems Journal, 5(1):12–29, 1966. (p 1)