

Dynamic Memory Model Reconfiguration in DEIMOS

Michael Clarke and Geoff Coulson

Distributed Multimedia Research Group,
Department of Computing,
Lancaster University,
Bailrigg, Lancaster,
LA1 4YR, U.K.
E-mail: [mwc, geoff]@comp.lancs.ac.uk

Abstract

Operating system design has traditionally followed a philosophy in which the system is structured as a fixed set of abstractions and mechanisms. This approach, however, is now showing its limitations in the face of new application areas; particularly those which demand extensibility and configurability. In this paper, we describe aspects of the design of a dynamically extensible operating system called DEIMOS. DEIMOS is unique in that it does not define a kernel entity. Instead, both traditional kernel abstractions and application specific services are encapsulated as ‘modules’ which can be loaded, configured and unloaded on demand (i.e. at run time) by a base system component called the Configuration Manager (which can itself be unloaded). The lack of a kernel gives DEIMOS great flexibility as applications have complete freedom to tailor their execution environment in accordance with their particular needs. Furthermore, applications can dynamically adapt their environment on an ongoing basis and the differing needs of diverse applications can, in many cases, be met simultaneously. The paper discusses the architecture of DEIMOS and focuses specifically on our approach to memory management including a description of how DEIMOS can support a variety of dynamically instantiable memory models (e.g. segmented memory, paged virtual memory, flat unprotected memory or software protected memory). Additionally, we discuss the possibility for combinations of these memory models to simultaneously coexist given the right conditions.

1. Introduction

DEIMOS (Dynamically Extensible Incrementally Modularised Operating System) is an enabling technology for the production of bespoke application execution environments. Unlike conventional operating systems (OSs), DEIMOS does not prescribe standard services such as file systems and device drivers nor even fundamental abstractions such as processes and memory models. Instead, it empowers system managers with the ability to specify and incorporate, at run time, only those services and abstractions they actually require. The role of DEIMOS is to act both as a ‘meta-service’ for application specific service introduction and as an arbiter for the many possible conflicts that can occur in this process.

DEIMOS is particularly applicable in application domains which require high levels of dynamically adaptive behaviour¹ [Clarke,98], for example;

- *mobile systems* which need to be dynamically adaptive to deal with varying levels of connectivity, power and resources [Blair,97], and
- *multimedia systems* which also have strong requirements for adaptivity and dynamic loading of components such as media encoders, media specific protocols and device drivers.

The DEIMOS concept is motivated by our belief that traditional OS architectures are unable to provide

¹ DEIMOS is also applicable in situations where custom, but less dynamic configurations are required; e.g. highly available systems which must tolerate service upgrades without down time, embedded systems, network computers and transaction processing systems (for instance requiring specialised disk management and caching functionality).

the degree of flexibility increasingly required by such application domains. Operating systems are traditionally architected as a number of tightly coupled managers exporting a static set of abstractions to the application via an application programmer interface (API). Although this architecture has worked well for decades, its weaknesses are now becoming evident in a number of areas. Firstly, such an architecture is hard to adapt for application domains such as the above which post-date the original OS design. Secondly, OS services are inevitably sub-optimal for any given domain because of their generality. Thirdly, OS services rarely utilise hardware specific facilities for extra performance because of the need for portability and backward compatibility.

For these reasons we believe that the original purpose of the traditional operating system, i.e. to make application production more convenient through generic service provision, has not only been eroded but is also becoming a limiting factor in the support of novel, high performance, application domains. Our argument is not that existing operating systems are unable to provide appropriate services; rather it is that the *usefulness* of such services is reduced when they are deployed in conventional operating system structures. It is the inability to adequately *manage* OS services which gives rise to the static system configurations seen today and leads to the inadequacies described above. To remedy this situation, we propose an *application-centric* approach in which application designers are responsible for specifying the services they require and the OS is then responsible for enabling and managing the inclusion of those services within itself.

Converting non-fundamental system services such as file systems, device drivers and timers into application selectable entities is a relatively straightforward task because much of the code making up these services is self contained and can execute with user level privileges ([Rashid,89] and [Liedtke,96]). However, more care is required when attempting to factor out fundamental abstractions such as the memory model, the concurrency model or the IPC. Without careful management, there is great potential for *conflict* between like-instances of such abstractions which may be required to coexist at run time for maximum flexibility. For instance, one application may wish to execute in a flat memory environment whilst another may wish to simultaneously execute in a paged memory environment. Additionally, such abstractions (particularly the memory model) often have wide ranging *side effects* which can potentially impact any other area of the system.

In this paper we describe the overall architecture of DEIMOS and then focus on the provision of configurable memory models in the DEIMOS environment. Memory models are particularly demanding in terms of conflict and side effect management and therefore their successful accommodation should provide convincing evidence for the feasibility of the DEIMOS methodology.

The rest of the paper is structured as follows: Section 2 reviews related work and presents a taxonomy of extensible operating system research. Section 3 then provides an overview of the DEIMOS architecture. Section 4 describes DEIMOS's memory management infrastructure in detail. Section 5 briefly discusses our implementation approach and, finally, section 6 offers some concluding remarks.

2. Related work

A fairly large body of research has addressed the need for extensible and configurable operating systems in the recent past. It is possible to classify this research as follows;

- *build time extensible systems* are configured for different application purposes by system administrators who select appropriate modules for the bootable kernel instance off-line. Examples include the latest Chorus microkernel [Abrossimov,96] and μ Choices [Tan,95],
- *library extensible systems* define a minimal kernel interface which provides a fixed primitive execution environment (threads and memory protection) plus wrappers over hardware services (which, for example, protect against contention and race conditions). Applications select and link at compile time to a *library operating system* which builds appropriate semantics on top of these policy-neutral kernel services. The prime example of this approach is the Exokernel project [Engler,95]. Others include Lipto [Druschel,93] and the Cache kernel [Cheriton,94],
- *reflectively extensible systems* are object oriented systems which define a *meta object protocol* (MOP) on system objects which can be used to modify aspects of the objects' implementation or

environment at run time. Such systems allow, in principle, the dynamic reconfiguration of the entire system for application benefit. Apertoss (now AperiOS) [Yokote,92] is the prime example of this type of system,

- *dynamically extensible systems* allow applications to download code into the system address space. This code can then access privileged (but secure) kernel resource interfaces to perform fast, application defined actions. To prevent downloaded code from executing arbitrary privileged instructions, compiler based protection techniques are used. Classic examples of dynamically extensible systems are SPIN [Bershad,95] and VINO [Seltzer,94].

Build time extensible systems do not allow applications to extend system services beyond boot time and are thus not directly related to our concerns. Library extensible systems permit service extensibility at link time but applications cannot tailor their execution environment as they execute. Reflective systems allow full, system-wide extensibility at run time and present a single abstraction for extensibility, but typically suffer from poor performance [Itoh,95] (mainly due to their heavy use of interpreted code). Finally, dynamically extensible systems allow applications to tailor services at run time through downloaded compiled code which cannot harm the static portion of the kernel, thus making the approach both safe and efficient. Unfortunately, there seem to be inherent difficulties in opening the *whole* system to dynamic modification. This means that much kernel machinery remains static and is therefore forced upon applications just as in traditional operating systems.

DEIMOS is a hybrid of the three categories of dynamic extensibility described above. It features a single unit of extensibility (the *module*), run time service tailorability, compiled code and is unique in not prescribing *any* kernel abstractions or mechanisms. In our approach, it is the responsibility of the system manager or even the application designer to configure the necessary system services and abstractions from an extensible pool of available modules as they are needed. Essentially, our goal in designing DEIMOS is to explore the implications of pushing the provision of flexibility to its absolute limits.

3. Architectural overview

3.1 Object model

DEIMOS adopts a mature and well understood object-based model for the structuring of its system services and applications; this is the computational model of the Reference Model for Open Distributed Processing (RM-ODP) [ITU-T,95] shown in figure 1. Although this model was designed primarily for distributed computing environments and is usually deployed in middleware platforms, we have found it equally applicable in the operating system domain.

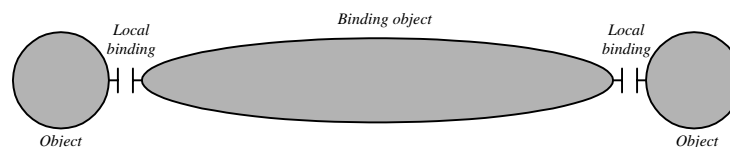


Figure 1: The RM-ODP computational model. Objects are encapsulated entities with multiple interfaces which communicate with other objects via binding objects. The recursion inherent in this model is terminated by the provision of system provided primitives called local bindings¹.

The fact that bindings are themselves objects and can thus be created and manipulated in the same way as any other object has a number of important benefits. One benefit is that the set of available interaction modes (e.g. messaging, request / reply, streaming) is extensible. In addition, binding objects can subsume resource allocation and thus may embody varying qualities of service.

A variety of base mechanisms such as traps, calls, messages, events, etc. can be used to implement binding objects and complex communications services can be built by composing or subclassing existing

¹ As local bindings are always intra-address space, they can be very efficiently implemented as machine level indirect CALLs.

binding classes to produce new ones.

In applying the RM-ODP computational model to DEIMOS, we adopt the term *module* in preference to the term ‘object’ to avoid confusion between the system level entities employed in DEIMOS and the language level entities commonly implied by the term object. DEIMOS uses four types of module as follows;

- *application modules* which implement applications,
- *system modules* which embody system services and abstractions. Examples are *memory modules* (described in detail in this paper), *concurrency modules* (which provide execution abstractions such as threads), *file systems* and *device drivers*¹,
- *binding modules* provide communications services between interfaces of the other three types of module; different communication mechanisms (e.g., see above) are encapsulated in different binding module classes; the interfaces of binding objects are attached to the interfaces of to-be-bound objects via primitive *local bindings*,
- *wrapper modules* give access to low level hardware resources (cf. library extensible systems; see section 2). It should be noted that these modules do not define policies over how the hardware is used; i.e. once modules have successfully bound to a wrapper interface, they may directly control the hardware as they wish. There are three types of wrapper module; *device access wrappers* offer a low level interface to the memory mapped IO address space of a device, *privileged instruction wrappers* allow applications to issue privileged instructions such as those for IO and halting the CPU, and *neutral service wrappers* encapsulate access to critical hardware resources (primarily memory management hardware and interrupt vectors). The *neutral memory manager* (NMM) wrapper module is described in detail in section 4.

Typically, application programmers will provide application modules while vendors and third party service creators will provide system and binding modules. In addition, the competent application designer is of course at liberty to hand craft system and binding modules as required.

3.2 The configuration manager

The *configuration manager* (CM) is a component of every DEIMOS boot archive and is responsible for the (application initiated) management of DEIMOS modules. To do this, it maintains a *system graph* (see figure 2) which represents all currently active modules together with their binding topology. Although it is a key component in the architecture, the CM is *not* a kernel in the traditional sense of the term. In particular, it is possible for the CM itself to be unloaded from the system if desired (e.g. if memory is in short supply). This will, however, result in a system that cannot be further reconfigured at run time.

The CM plays two key roles within DEIMOS. Firstly, it is responsible for overseeing the loading, binding, rebinding and eventual deletion of local bindings between module interfaces. Requests for these actions can be made to the CM from arbitrary modules but are usually initiated by application modules in the first instance. Secondly, the CM is responsible for identifying and policing the *contention* that inevitably arises when different applications make conflicting demands of the available system resources via the choice of system modules that they bind into the system graph.

The CM polices contention by checking *protection constraints*² associated with each interface involved in a local binding request. These consist of i) a set of *<name, value>* pairs and ii) a set of predicates which are composed of further *<name, value>* pairs, standard boolean connectives and built in functions. When asked to bind two interfaces I_1 and I_2 , the CM evaluates the predicates from I_1 using values from the pairs of I_2 and evaluates the predicates in I_2 using values from the pairs of I_1 . The local binding is only created if both evaluations return true.

¹ System policies such as scheduling and paging policies etc. can also be encapsulated as system modules so that they are visible and therefore subject to change.

² These are an extension of the notion of *environmental contract* in RM-ODP.

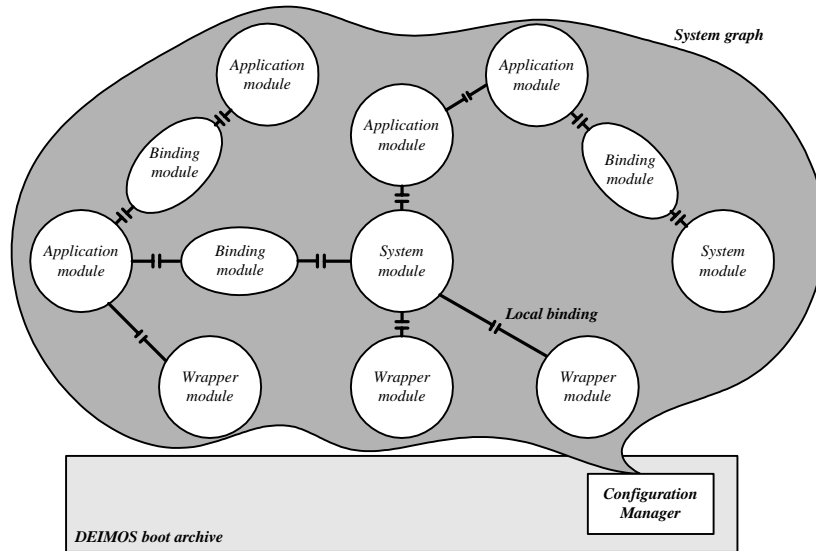


Figure 2: The system graph is maintained by the Configuration Manager and is made globally visible to applications as a read-only data structure. The figure illustrates an example configuration consisting of all four types of module with local binding interconnections.

Examples of built in functions are as follows (the first two functions depend on unique names associated with modules in the form of a `<moduleName, string>` pair; modules must supply a pair of this form to participate in these functions, otherwise these functions evaluate to false);

- `before(<moduleName, "x">, <moduleName, "y">)` returns true if an attempt is being made to bind the associated interface to a module other than module *y* or if the interface is already bound to module *x*,
- `compatible(<moduleName, "x">, <moduleName, "y">)` returns true if the associated interface is not currently bound to either module *x* or module *y* or if the associated interface is currently bound to module *x* and an attempt is being made to bind to module *y* or if the associated interface is currently bound to module *y* and an attempt is being made to bind to module *x*,
- `wildcard()` returns a 'special' value that matches any value for a given name; e.g. `<mem_requirement, wildcard(>` matches any pair with the name `mem_requirement`,
- `bindcount()` returns the number of bindings in which the associated interface is currently involved.

As an example of the protection constraint checking procedure, consider three interfaces I_1 , I_2 and I_3 with the pairs and predicates illustrated in figure 3. I_2 is already locally bound to I_1 and an attempt is being made to locally bind I_3 to I_1 . Firstly, the predicates of I_3 clearly match the pairs of I_1 . The case of the other required match (that of the predicates of I_1 and the pairs of I_3) is more complex. In this case, despite the fact that I_2 is already bound to I_1 , the `compatible()` term permits the simultaneous binding of I_3 to I_1 . Note that the proposed binding would have been disallowed if, for example, an interface with `moduleName segmented_mem_model` had previously bound to I_1 (in fact, `segmented_mem_model` is not compatible with anything else in this example). Note also that the present constraints would equally have permitted the ordering of the two bindings to have been I_3 -before- I_2 rather than I_2 -before- I_3 (the `before()` function could have been used instead of `compatible()` to enforce some particular ordering). In effect, the above protection constraints are allowing any *single* memory module to be bound to I_1 but only `flat_mem_model` and `paged_mem_model` may be simultaneously bound.

Finally, it must be noted that although protection constraints are necessary to constrain the evolution of the system graph, they may not be sufficient in an environment of mutual distrust. For instance, applications initially share a flat unprotected address space; in such an environment there is nothing to prevent modules from directly accessing other modules to which they are not legally bound. Applications

must explicitly load a memory model with appropriate protection characteristics to prevent such circumvention and force interactions to be channelled through local bindings that have been validated by the CM.

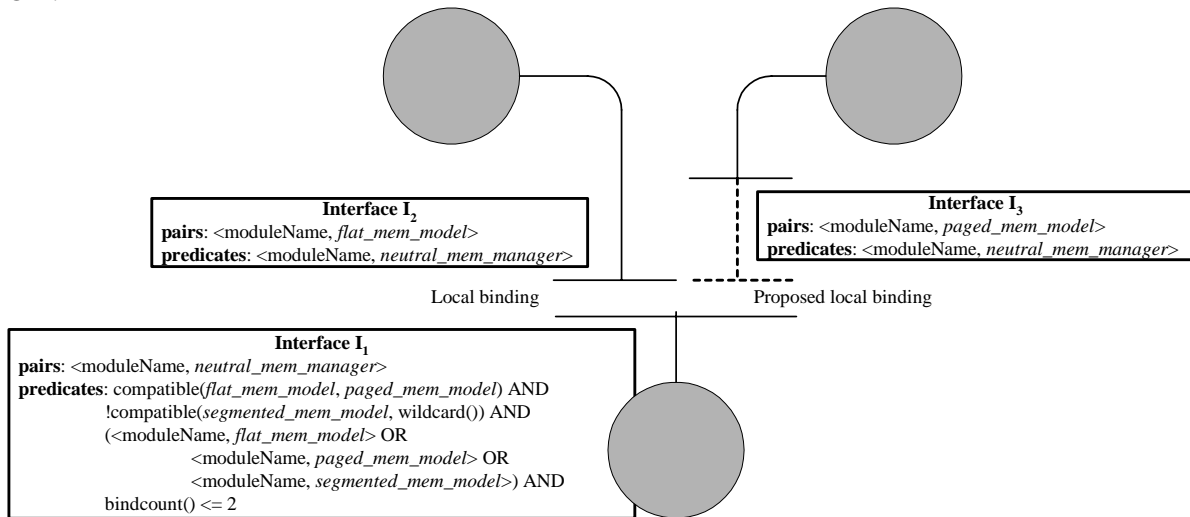


Figure 3: A scenario illustrating the use of protection constraints.

3.3 The DEIMOS core

The DEIMOS *core* (see figure 4) is an activated boot archive which consists of a bootstrap sequence, the CM, a default loader, optional disk and network drivers, a set of wrapper modules and an *initial user module* (IUM). The latter is administrator selectable at build time and can perform arbitrary actions (cf. the UNIX *init* process). The core is protected by the initial hardware configuration so that standard application, binding and system modules may not circumvent access restrictions (i.e. modules must establish CM arbitrated bindings between themselves and wrapper modules before they can access the underlying wrapped hardware).

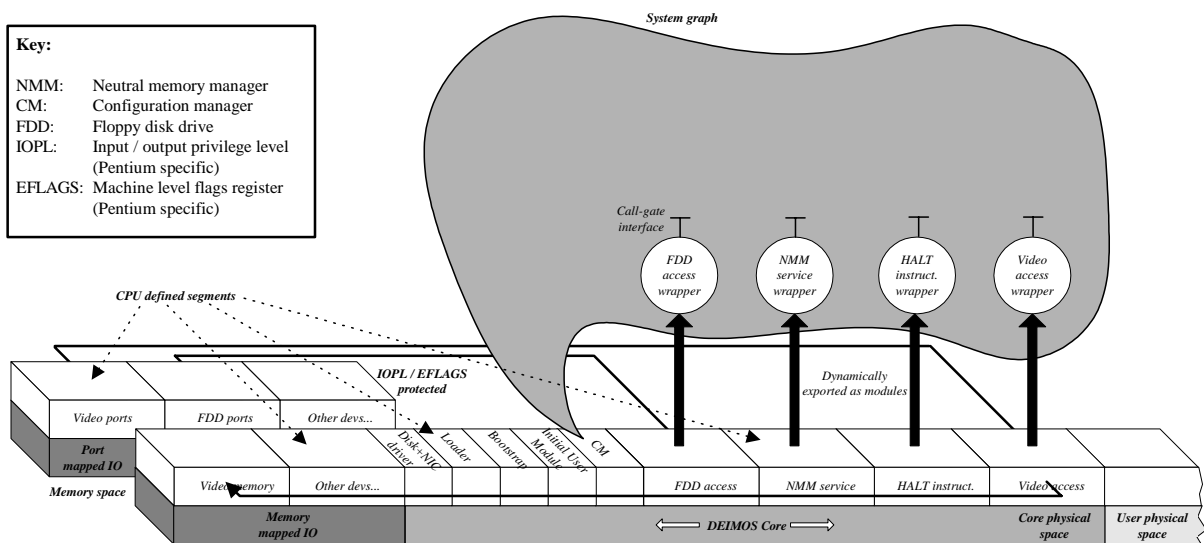


Figure 4: The DEIMOS core resides in a dedicated protected segment and consists of the bare minimum of functions required to bootstrap the system. The remainder of free memory, known as the User Physical Space (UPS), is available for use by subsequently loaded modules and is initially structured as a single, shared, unprotected segment. The figure shows the initial system deployment

including the IUM and a selection of wrapper modules with associated memory mapped IO address spaces and IO ports illustrated as being under their control. In the case of our Pentium [Intel,95] based prototype, these memory regions are also protected from direct module access by machine level mechanisms (i.e. IOPL and EFLAGS).

Following completion of the DEIMOS boot sequence, it is the responsibility of the IUM to build a workable execution environment by loading and configuring (via the CM) an appropriate set of system modules. One key function of these modules will be to complete the hardware initialisation process that was started by the bootstrap code. This is achieved by binding to the appropriate wrapper module, e.g. the neutral memory manager in the case of memory modules, and establishing an appropriate configuration. Note that it is perfectly possible to change this configuration at a later point in the evolution of the system by loading and binding replacement modules or even by adding new modules alongside the initial set. On the other hand, it is also perfectly possible to preclude or constrain such behaviour by associating appropriate protection constraints with the interfaces of the initial module set.

4. Memory management in DEIMOS

4.1 Architecture

In DEIMOS, a memory model is made up of two architectural components. The first is the *neutral memory manager* (NMM). This is a wrapper module that is included in the DEIMOS core. The second comprises one or more system modules known as *memory modules* which bind to and use the primitive services of the NMM to build the desired memory model semantics. Note that both the NMM and the memory modules that use them are necessarily platform dependent as they need to manipulate the underlying hardware.

4.2 The neutral memory manager (NMM)

4.2.1 General Considerations

The NMM is responsible for protecting the underlying memory management hardware of a given platform from direct module access. This hardware is selectively exposed at the lowest level, i.e. in terms of machine dependent data structures, registers and instructions, to modules which bind to the NMM's service interface. In addition, the NMM provides a number of *value added* services, abstractions and internal mechanisms which build on the underlying hardware to enable more convenient manipulation of memory. Although these value added services combine several elements of memory management functionality we are careful to ensure that they do not impose memory management *policy*; the latter is strictly the concern of application supplied memory modules.

Although some functionality, which we term semi-portable, can be expected to be common to each implementation (e.g. the support of paged virtual address spaces), most of the NMM implementation and interface will be highly machine dependent. This is a natural consequence of our premise that current operating systems do not attempt to fully utilise the underlying features of a given platform and thus restrict applications from reaching their full performance potential. Our system enables such low level optimisation at the cost of the increased development effort needed to implement memory models across platforms (compared with other types of system module). We expect that third party module suppliers would take most of the responsibility for developing platform specific memory module implementations. Intelligent design would ensure that memory modules contain as little, and as well isolated, NMM dependent code as possible in order to ease their cross platform porting.

4.2.2 A Pentium specific NMM

Our Pentium based NMM prototype manages two major classes of hardware address space: *physical segment address spaces* (PSAS), which are defined as contiguous regions of physical memory bounded

by a CPU defined segment, and *virtual segment address spaces* (VSAS), which are similar except that they refer to virtual memory. PSASs are available whether paging is enabled or disabled, while VSASs are only available when paging is enabled. In addition to hardware address spaces, the NMM defines the notion of a *region* which is a contiguous section of memory existing within the context of a hardware address space¹. Regions can be realised in a number of different ways on the Pentium processor, e.g. as *local segments* or as software defined extents. Finally, for every hardware address space that is created, the NMM creates an associated *memory map* which describes the allocation of non-overlapping regions within each address space. Region allocations are requested by memory modules and overseen by the NMM itself which updates its memory maps accordingly.

Table 1 presents a selection of the functionality available from the service interface of our NMM prototype. It highlights the full spectrum of services ranging from those which are fully machine dependent (e.g. segmentation), to those which are semi-portable (e.g. paging functions) to those which are value added functions that enable convenient memory management for memory modules and pertain to the internal operation of the NMM.

<i>Machine dependent methods</i>	
CreateGlobalSegment()	Create a globally visible segment <i>only when paging is disabled</i> . This causes the instantiation of a new PSAS hardware address space and an associated memory map.
CreateLDT()	Create a local descriptor table to hold local segment descriptors for a single hardware task.
CreateLocalSegment()	Create a local segment whose descriptor is only visible from the hardware task which owns the LDT that it resides in.
CreateTSS()	Create a task state segment to enable hardware context switching.

<i>Semi-portable methods</i>	
Paging()	Enable or disable paging.
CreatePageTable()	Creates a page table for a new hardware task. This causes the instantiation of a new hardware address space, and subsequently, a corresponding memory map.
CreatePageMapping()	Create a virtual to physical page mapping in a page table.
InvalidatePageMapping()	Mark a virtual to physical page mapping as invalid.
PhysToVirt()	Perform a reverse lookup in a page table to find the virtual page that is mapping a given physical frame.

<i>Value added methods</i>	
CreateRegion()	Create a region within a pre-existing hardware address space.
QueryRegion()	Ascertains the allocation status of a region of memory with a given hardware address space.
IdentifyRegion()	Returns the identifier of the region that the caller currently occupies.

Table 1: A selection of the methods available in the Pentium NMM implementation. Note that the level of abstraction is significantly lower than that of even the internal interfaces of conventional OS kernels. Traditionally, this type of functionality is hard coded into start up sequences and is not available for interaction once the OS has initialised. However, in DEIMOS, this level of functionality must be continuously available so that memory modules can complete the hardware initialisation that was started by the DEIMOS bootstrap sequence in an application specific, ongoing and open ended way.

One interesting Pentium specific feature exploited in our NMM implementation is the possibility of dynamically altering the processors paging status. Using this feature, a system can initially establish a simple PSAS based memory model (no paging) without precluding the possibility of subsequently instantiating an additional VSAS based memory model (with paging). This situation may also arise in reverse, i.e. a pre-existing memory model may have enabled paging and a subsequently loaded model may wish to function purely within a PSAS environment. Because simply switching off paging in such a situation would disrupt modules supported by the original model, we provide a value added service to enable the transparency of regions with respect to paging status; i.e. the NMM guarantees that addresses

¹ The nature of a region, i.e. physical or virtual, depends on the type of hardware address space in which it is created.

in a region remain valid whether or not paging is enabled. As an example, consider the boot time situation, before paging is enabled, where there are two PSASs available in which regions can be allocated, i.e. the DEIMOS core space and the user physical space (UPS) (see figure 4). When paging is enabled, the NMM creates a custom VSAS with a number of internal regions which correspond to the original physical segments (i.e. their virtual addresses are mapped to the original physical addresses through page tables). In addition, a mapping is established in the NMM between the identifiers of the original PSASs and the identifiers of their now corresponding VSAS. Therefore, when an attempt is made to allocate a region from a PSAS in the original UPS, this mapping translates the request to allocate a corresponding virtual region in the custom VSAS. Note that the default action of the NMM is to transparently back all requests for virtual memory with an equal amount of physical memory *unless* the memory module making the request explicitly indicates that it will handle its own physical allocations. This would occur, for example, with a memory module implementing a paged memory model (see section 4.3.2) and would require the use of other NMM services in order to manually update the system page tables with the correct virtual to physical translations.

4.3 Memory modules

4.3.1 General Considerations

It is the role of memory modules to implement particular flavours of memory organisation on top of the primitive services provided by the NMM. In our Pentium based prototype, examples of memory organisations are flat, segmented, paged and combined segmentation with paging. We additionally support *software memory models* whose client modules can only exist in the context of a pre-existing hardware address space (which itself conforms to one of the above hardware organisations). Software memory models use software techniques, originally devised by the compiler and language communities, to mutually protect separate client modules. Examples of these techniques include software fault isolation [Wahbe,93], safe languages, e.g. Modula-3 [Bershad,95], proof carrying code [Necula,96] and interpreted environments [Sun,97].

The primary purpose of a memory model is to export the ability to create address spaces for module execution. Beyond this, a memory model may offer the typical application memory management functions usually found in the API of a conventional operating system. Examples include the creation of extra mapped regions for dynamic data and the ability to share memory with other applications.

Memory models cannot operate in isolation from other system modules because the effects of memory organisation permeate all aspects of both system and application execution. In addition, memory management is only one aspect of a complete computational environment. To implement a complete environment, complex dependencies and coupling must exist between memory modules, concurrency modules, loader modules and the NMM. To shield this complexity from application modules, it is potentially useful to define *execution modules* which export a common API representing some particular execution model in terms of loading, memory organisation and concurrency functions¹ (see figure 5).

4.3.2 An Example Memory Model

We now describe, as an example, how one particular type of memory model - a paged virtual address space model with one level store - can be installed and configured in the Pentium environment. We also demonstrate how the model subsequently offers its services to other modules.

When DEIMOS completes its boot strap sequence, control is immediately transferred to the initial user module (IUM) which, using the services of the CM, binds to the default system loader to orchestrate the loading of the paged memory model (which we here assume to be implemented as a single memory module). The default system loader avoids the bootstrapping problem of finding memory for the memory module by working in conjunction with the NMM to allocate an appropriately sized region for the memory module from the UPS. Once loaded, the memory module executes its own initialisation routine.

¹ Execution modules are not dissimilar to OS personalities in a microkernel environment.

This involves binding to and interacting with the NMM, i) to populate the initial page directory and page tables with appropriate mappings to transform any existing physical segments that contain non-core modules into a single VSAS (see section 4.2.2), ii) to initialise a free list of page frames based on the NMM's remaining PSAS memory maps (i.e. the core and UPS) and iii) to switch on paging. After the memory module has initialised, control returns to the IUM which will typically load and locally bind appropriate concurrency and execution modules.

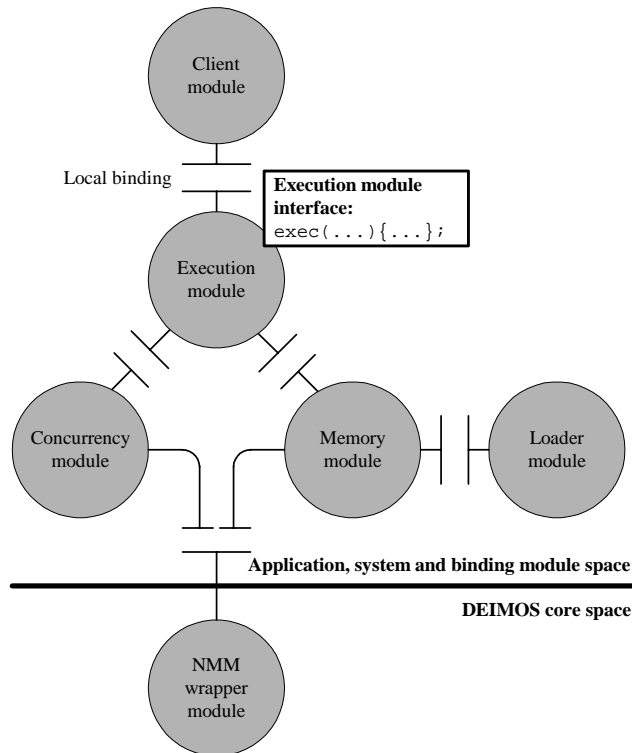


Figure 5: An example configuration of a complete computational environment. The execution module exports an `exec()` style method in its service interface which allows client modules to load and run new modules. Coupled to the execution module are particular flavours of concurrency and memory model. The latter requires the services of a loader to actually load module images from a disk or the network. Both the memory and concurrency modules make use of the NMM's service interface because, on our Pentium implementation, hardware context switching data structures are an integral part of the hardware memory management data structures.

As part of their initialisation, execution modules bind to the service interfaces of the memory and concurrency modules. Using these interfaces, execution modules can implement the autonomous (from the IUM's perspective) sub-loading of client modules. A typical procedure for loading a module via an execution module is as follows;

- the execution module requests the creation of a VSAS and an appropriate region through the memory module's service interface (which in turn contacts the NMM to carry out the request),
- the memory module is then tasked with loading the desired module's image into this space using its own loader(s),
- finally, the execution module contacts the concurrency module to associate the newly loaded module with a new flow of execution.

In normal operation, the paged memory module will receive client requests for the allocation of physical memory to back faulting or new regions of virtual memory. To satisfy these requests, the

memory module consults its free physical list and asks the NMM to verify the status of the selected pages(s). If the requested frame is not available (e.g. it has since been allocated to another memory model by the NMM; see section 4.3.3), an error code is returned. In such a case, the memory module asks the NMM to perform a *reverse lookup* to locate the page table entry that is currently mapping the frame. The frame can then be flushed to backing store in an operation that invalidates the page table entry and causes an update in the NMM's memory map of the frames corresponding PSAS. The now free frame can be used to satisfy the original request to back some arbitrary region of virtual memory (via further updates to the page table structures). A similar scenario applies when the paged memory module's free list is empty. In this case, it determines which frame(s) to flush by inspecting paging attributes such as read/write, access count, dirty bit etc. which can be obtained via further methods on the NMM's service interface.

4.3.3 Coexistence of memory models

To achieve maximum flexibility and generality, it should be possible to simultaneously support multiple memory models in the same DEIMOS instance. An important requirement in such situations is that pre-existing memory models should continue to function (by exporting their services and maintaining client module execution environments) even in the face of radical changes to memory organisation instantiated by more recently loaded models.

Although there are clearly cases where particular memory models cannot co-exist, co-existence does turn out to be possible in a surprising number of cases (e.g. [Clarke,99] has discussed the coexistence of paged, segmentation and flat memory models). Incompatibilities are expressed in terms of protection constraints on the interfaces of the modules concerned and are policed at bind time by the CM as described in section 3. To maximise the potential for co-existence, memory modules should be written in such a way that their initialisation routines take full account of the memory organisation into which they are being loaded (typically by referring to the system graph to ascertain which memory models already exist).

5. Implementation Approach

Our prototype implementation of DEIMOS is based on the Chorus CLASSIX OS [Chorus,96] running on Pentium PC hardware. CLASSIX is a build time extensible operating system which provides a number of facilities useful in implementing the DEIMOS concept. These include: i) dynamically executable processes that run in supervisor space, ii) the ability to attach arbitrary code to interrupts so that hardware specific modules and trap based bindings can be implemented and iii) memory management selection which allows a (build-time) choice between the alternative memory models supported by PC hardware. In addition, we exploit the CLASSIX *hot restart* technology; this feature, which is based on Chorus port migration, is useful for dynamically introducing new services in DEIMOS.

Many components in the DEIMOS core are extensions of core CLASSIX modules and are thus available with little development effort. The CM is an exception to this; it is implemented as a scratch built boot actor, i.e. a process started during the CLASSIX boot process. In further developing the DEIMOS concept, our strategy is to progressively disable more and more elements of the CLASSIX development environment thus allowing DEIMOS modules to take progressively greater responsibility for dynamically managing system resources. Essentially, we are incrementally bootstrapping the DEIMOS environment from the existing CLASSIX environment. Eventually, DEIMOS will exist as a self contained system and CLASSIX will no longer be required.

6. Conclusions

DEIMOS is a minimally prescriptive framework for the construction of dynamically extensible operating systems. The framework is based on a well understood object-based computational model and uses a configuration manager to track and control the dynamic evolution of the system in terms of its constituent modules. DEIMOS is unique in that it defines no kernel entity. Thus it manages to be non

prescriptive even in terms of such fundamentals as choice of IPC, concurrency and memory model.

This high degree of flexibility makes DEIMOS suitable for all classes of computing in which dynamic configurability is an important requirement. Other benefits of the design include a possible decrease in time in the development cycle. For example, memory models can be dynamically introduced for debugging rather than having to continually re-specify, rebuild and reboot the kernel as is necessary on most systems, and device drivers can be loaded into isolated and unprivileged address spaces and then migrated on-the-fly to spaces which enable optimal performance.

The benefits of DEIMOS's memory management architecture can be summarised as i) the ability to optimally exploit hardware specific facilities, ii) the ability to simultaneously support applications with diverse requirements on the same system and iii) the ability to dynamically reconfigure for the optimal use of resources when competing modules are supported in the same system. Our approach to memory management gives applications the freedom to install memory models optimally tailored to their particular requirements. DEIMOS achieves this by performing only the minimum amount of hardware initialisation required to support applications and then allowing application defined memory models to complete this initialisation in an open ended way. Even when a memory model has completed this initialisation process, there is still the capability to reconfigure the memory management hardware dynamically (e.g. to simultaneously support more than one memory model) if application needs change.

References

- [Abrossimov,96] Abrossimov, V., Boule, I., Germond, J.J., Lipkis, J., Rozier, M., Ruget, F., Valette, E., "STREAM-v2 Kernel Architecture and API Specification Version 1.0". Technical Report ST/TR-94-2.6 (Under Esprit Project 8305), Chorus Systems, France, 1996.
- [Bershad,95] Bershad, B.N., Savage, S., Przemyslaw, P., Sire, E.G., Fiuczynski, M.E., Becker, D., Chambers, C., Eggers, S., "Extensibility, Safety and Performance in the SPIN Operating System". Proceedings of the 15th ACM Symposium on Operating Systems Principles, pp 267-284, Copper Mountain CO, U.S.A., December 1995.
- [Blair,97] Blair, G.S., Coulson, G., Davies, N., Robin, P. and Fitzpatrick, T., "Adaptive Middleware for Mobile Multimedia Applications", Proceedings of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97), St. Louis, MI, U.S.A., May 1997.
- [Cheriton,94] Cheriton, D.R., Duda, K.J., "A Caching Model of Operating System Kernel Functionality". Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation (OSDI '94), Monterey CA, U.S.A., November 1994.
- [Chorus,96] Chorus Systems., "Chorus CLASSIX r3 Technical Overview". Technical Report CS/TR-96-110.12, 1996.
- [Clarke,98] Clarke, M., Coulson, G. "An Architecture for Dynamically Extensible Operating Systems". Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDs'98), Annapolis MD, USA, May 1998.
- [Druschel,93] Druschel, P., "Efficient Support for Incremental Customisation of OS Services". Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems, pp108-111, Asheville NC, U.S.A., December 1993.
- [Engler,95] Engler, D.R., Kaashoek, M.F., O'Toole jnr, J., "Exokernel: An Operating System Architecture for Application-Level Resource Management". Proceedings of the 15th ACM Symposium on Operating Systems Principles, pp 251-266, Copper Mountain CO, U.S.A., December 1995.
- [Intel,95] Intel Corporation., "Intel Pentium Processor Family Developers Manual, vol 3: Architecture and Programming Manual", Order number 241430, Intel Corporation, Santa Clara CA, U.S.A., 1995.
- [Itoh,95] Itoh, J., Lea, R., Yokote, Y., "Using Meta-objects to Support Optimisation in the Apertos Operating System". Technical Report SCSL-TM-95-006, Sony Computer Science Laboratory Inc., Japan, 1995.
- [ITU-T,95] ITU-T, ISO/IEC Recommendation X.902, International Standard 10746-2, "ODP Reference Model: Descriptive Model", January 1995.
- [Liedtke,96] Liedtke, J., "Toward Real Microkernels". Communications of the ACM (CACM), 39(9), pp 70-77, September 1996.
- [Necula,96] Necula, G.C., Lee, P., "Safe Kernel Extensions Without Run-Time Checking". Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation (OSDI'96), pp 229-243, Seattle WA, U.S.A., October 1996
- [Rashid,89] Rashid, R., Baron, R., Forin, A., Golub, D., Jones, M., Julin, D., Orr, D., Sanzi, R., "Mach: A Foundation for Open Systems". Proceedings of the 2nd Workshop on Workstation Operating Systems (WWOS2), September 1989.
- [Seltzer,94] Seltzer, M., Endo, Y., Small, C., Smith, K., "An Introduction to the Architecture of the VINO Kernel". Harvard University Computer Science Technical Report 34-94, 1994.
- [Sun,97] Sun Microsystems, "JavaOS: A Standalone Java Environment". <http://www.javasoft.com/products/javaos/javaos.white.html>, February 1997.
- [Tan,95] Tan, S-M., Raila, D.K., Campbell, R.H., "An Object-Oriented Nano-Kernel for Operating System Hardware Support". Proceedings of the 4th International Workshop on Object-Oriented in Operating Systems, Lund, Sweden, August 1995.
- [Wahbe,93] Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L., "Efficient Software-Based Fault Isolation". Proceedings of the 14th ACM Symposium on Operating Systems Principles, pp 203-216, Asheville NC, U.S.A., December 1993.
- [Yokote,92] Yokote, Y., "The Apertos Reflective Operating System: The Concept and Its Implementation". Proceedings of the ACM Object Oriented Programming Systems, Languages and Applications'92 (OOPSLA'92) Conference, pp 414-434, Vancouver BC, Canada, October 1992.