APM

POSEIDON HOUSE • CASTLE PARK • CAMBRIDGE CB3 0th UNITED KINGDOM

+44 1223 515010 • Fax +44 1223 359779 • Email: apm@ansa.co.uk • URL: http://www.ansa.co.uk

ANSA Phase III

FlexiNet - Extensible Kernel Investigation

Oyvind Hanssen

Abstract

In this report we investigate what is the main design issues and options in extensible operating systems, and how those issues are being addressed in some of the research prototypes that are being developed. Design issues like kernel architecture, protection, and conflict resolution are discussed. We also look at how object orientation and reflection may support the design of customisable systems. The focus is mostly at dynamic extensibility and an important issue here is whether extensions should be compiled or interpreted. Java seems to offer a compromise, especially when offering just-in-time compiling for critical code-parts.

APM.2002.01.00	**Approved** Technical Report	03 July 1997

Distribution:
Supersedes:
Superseded by:

TABLE OF CONTENTS

1 INTRODUCTION

As computers are becoming more interconnected and mobile, and as multimedia computing is emerging, it is desirable to make operating systems more customisable, i.e. applications/users are given more control over implementation issues or non-functional aspects, by being able to add or change parts of the O.S. There is a growing research interest in customisable and extensible operating systems, and what motivates this interest include the following:

- The need for performance optimisation by using the right algorithms/resource management policies in the right place. For instance, a well known problem is that caching or paging strategies may not fit the application's usage patterns. E.g. the least recently used page replacement strategy (LRU) used by many operating systems is suboptimal for many applications, especially database management systems[1].
- The need for functionality extension - it is unreasonable to expect any one system to possess all of the functions needed by all applications. For instance in file systems, one may add compression, replication, etc.
- Applications or O.S. extensions should be able to reuse kernel code (with the necessary specialisations) instead of duplicating it just because it isn't available or tuned for the particular application.

No generic operating system can ever satisfy all applications; it is important to permit individual applications to customise the system by specifying or providing specialised policies.

1.1 Focus

When studying customisability (i.e. users/applications ability to change the systems behaviour to its needs), we focus at extensible operating systems, i.e. operating systems that are based on a microkernel which implements a minimal set of functionality which every instance of the O.S. have. Operating system services are extensions to the microkernel that may be replaced, added or removed by users/applications. Less flexible (but still useful) approaches to customisability may e.g. be to provide a set of pre-defined policies the user declaratively can choose from.

[1] Database Management System implementers are often forced to bypass the paging subsystem and implement their own paging.

1.2 Goals

The goals of this report is to investigate what is the main design issues and options in extensible operating systems, and how those issues are being addressed in some of the research prototypes that are being developed. We also look at how object orientation and reflection may be useful in constructing customisable systems.

1.3 Report summary

In section 2, we go through some important design issues for extensible operating systems: First, extensibility may be static (compile time customisation) or dynamic (run-time). Dynamic extensibility may be accommodated by either the traditional microkernel approach, where extensions are implemented in user-space, or the extensible kernel approach, where they may be inserted or replaced in the kernel by applications.

Another issue is how to prevent erroneous extensions from corrupting the rest of the kernel. Protection against illegal memory access is done by hardware (the microkernel approach) or by software. Another problem is resource monopolising which may be tackled by detecting the problem and killing the offending extension.

Other issues include the persistence of extensions, extension granularity and the resolving of conflicts between extensions.

In section 3, we discuss the role of object orientation in construction of customisable operating systems and look at a couple of operating systems prototypes that investigate reflection. A problem is to give applications control over implementation decisions without sacrifying the benefits of abstraction. A promising approach to this is to expose implementation issues and non-functional aspects, where needed, but separate from the functional interfaces. Those aspects are controllable by applications through meta object interfaces.

We illustrate the concepts discussed in section 4, by presenting two prototype operating systems that take different approaches to customisability and extensibility, namely Nemesis and μChoices. We don't examine all related research in this paper, just a illustrative selection.

1.4 Acknowledgements

The author of this report is seconded to the ANSA Phase III programme by the University of Tromso and is supported by a NATO Science Fellowship through the Norwegian Research Council grant no. 116590/410.

Thanks to Andrew Herbert, Billy Gibson, Richard Hayton and Zhixue Wu for valuable advice and comments.

2 DESIGN ISSUES FOR EXTENSIBLE SYSTEMS

In this section, we give an overview of research in extensible operating systems, mostly focused at design issues. This is based on [Seltzer96] where we find a survey of the main design issues of extensible operating systems and some related research prototypes. The focus here is at extension technology, protection, extension persistence, extension granularity and conflict resolution.

2.1 Extension technology

A first step towards extensibility is to provide compile time options for configuring the resulting O.S. to particular needs. This is called static extensibility. Scout [Montz94] is an example of a toolkit that can be used to implement special purpose operating systems. Scout is designed primarily for networking applications and the abstraction of paths is central. It supports both non-realtime and soft-realtime applications. A step further is to support dynamic extensibility, i.e. behaviour can be added or changed at run-time. There are two different approaches to this: The microkernel approach and the extensible kernel approach.

The traditional microkernel approach (see e.g. [Accetta86]) means that O.S. functionality is moved out of the kernel and implemented as servers running in user space.

The traditional microkernel approach has a very high cost, because of the need for frequently crossing of protection boundaries. This is the main motivation of the extensible kernel approach. It means that applications may change the behaviour of the kernel itself, i.e. application functionality may be dynamically downloaded into the kernel and run there. Work taking this approach include SPIN [Bershad95], VINO [Small95] and to some extent, μChoices (see also section 4.2).

Exokernels [Engler95] may be regarded as a variant of the first approach. They are based on an idea of removing all abstractions from the kernel and implement them in user space, typically as libraries which are linked into each application process. The kernel is left with very few responsibilities like providing secure bindings between user-level operating systems and hardware. A prototype Exokernel, Aegis also allows downloading of code into the kernel to improve performance. Another example of an Exokernel is the V++ Cache Kernel [Cheriton94]. Here, the approach to performance

improvement is that active objects associated with basic operating system facilities are cached in the kernel.

Nemesis (see section 4.1) takes a exokernel-like approach. The system is organised as a set of domains (analogous to processes) which are scheduled by a very small kernel. Server domains (typically device drivers) are allowed, but as little as possible is done in server domains and as much as possible is done in application domains. Operating system functionality is mostly implemented as small library modules that may be dynamically loaded and linked into application domains.

2.2 Protection

There are two potential problems that the kernel needs to protect the system from. First from extensions that access (read, write or jump to) memory-addresses they weren't meant to access. Second from extensions that monopolise resources in an erroneous way.

2.2.1 Preventing illegal memory access

The traditional microkernel approach implies hardware protection since extensions are implemented as separate processes. Because, crossing protection boundaries is expensive (with respect to performance), this approach may have a significant performance cost.

Loading software into the kernel instead may therefore be an attractive alternative, especially when the extension communicates intensively with the rest of the kernel, but the problem is how to prevent buggy extensions from corrupting the integrity of rest of the kernel, by accessing memory or IO addresses they aren't meant to access. There are two main approaches to software protection:

- Compile-time protection: The compiler ensures that the generated extension code is safe. This is typically done by using safe languages, but unsafe languages like C can be used if the compiler (or a post-processor), prevents the generated code from accessing illegal addresses. This technique is referred to as "sandboxing".
- Run-time protection which can be either some kind of adding run-time checking to the extension code, or the use of interpreted languages which are interpreted in the kernel. The interpreter do safety checks.

Software protection is used in e.g. SPIN, that use a type safe language, Modula-3 to implement extensions and in VINO, where extensions are written in C++ and sandboxed, i.e. compiled by a trusted compiler that prevent generated code to access addresses outside its own address range, except for permitted entry points. VINO also runs extensions in the context of transactions. This supports recovery if extensions fail or must be aborted.

The problem of this approach is how the kernel can't be sure that the code is really generated by a trusted tool, when asked to download it. Some

cryptographic checksum scheme can be used, but this approach introduce new problems like key distribution and trust.

Another approach that aims to solve this problem is to use interpretative languages. μChoices allows downloading of software agents into the kernel, typically for aggregating multiple system calls and thereby minimising kernel/user space boundary crossings. These are implemented in a scripting language similar to TCL, but using Java is considered as the next step. Java is a safe language that is compiled to a intermediate form that is interpreted by a virtual machine which does additional safety checks in run-time. Java then offer both compile-time and run-time protection.

2.2.2 Preventing resource monopolising

None of the approaches mentioned above prevent badly behaving extensions from stealing all resources. To prevent this, extensions cannot be allowed to disable interrupts and the kernel must have a mechanism to detect the problem and kill the offending extension.

2.3 Extension persistence

Another issue is the persistence (or lifetime) of an extension. This is closely connected to the question what object the extension is bound to. There are two approaches to this.

- Extension is bound to application - It exists as long as the application runs.
- Extension is bound to resource - It exists as long as the resource it manages exists.

There is clearly a need for many extensions to persist longer than the application and to be shared between different applications, e.g. when it manages a long lived resources as a file system. This shouldn't hinder the extensions to evolve over time or even to be moved to another machine.

Related research approach this in different ways. Exokernels allows persistence in the sense that extensions can be implemented in shared libraries. Because extensions live in each application process, they cannot be directly bound to resources. In SPIN, extension may outlive the process that installed it and then becomes part of the running kernel. They don't survive reboots though. In VINO, extensions may be bound to both applications and files and may then be persistent, also over reboots.

2.4 Extension granularity

What is the unit of modification for extensions? We can distinguish between three levels of granularity:

- Module extensibility (coarse grain). Modules are replaced as a whole.

- Procedural extensibility (fine grain). Extensions can be defined in arbitrarily small units.
- Limited procedural extensibility. Extensions may be associated only with designated kernel entry points.

One example of module extensibility is the traditional microkernel approach. The unit of replacement is the entire server implementing it. Exokernels may define extensions in arbitrarily small units since the whole thing is implemented in the application process. Nemesis offer module extensibility, but the modules may be small, implementing one or more object-types. SPIN supports a variant of module extensibility, i.e. an extension can be implemented as subtype of another. In the VINO system, one is allowed to overload only specific methods of the objects that the kernel is constructed from. VINO is thereby supporting limited procedural extensibility.

2.5 Resolving of extension conflicts

Different extensions may issue conflicting resource or policy requests. To resolve conflicts, we either has to find a compromise that everyone can live with (some extensions can't have exactly what they requested) or some extension has to fail. Requests may be hard or soft. Hard request must be completely fulfilled or the requester must terminate. There are three types of conflict between extensions:

- Resource contention, The sum of the requests for a fraction of a resource (e.g. memory or network bandwidth), by the different extensions is more than what is available
- Policy contention. Differing (conflicting) policies (e.g. scheduling policies) are requested.
- Interference. Extensions require the same physical resources.

The only solution for interference is time multiplexing. If multiplexing cannot be done here, some extension must fail. An example is virtual memory, which may be multiplexed through paging or swapping.

An approach to resolving (soft) resource contention and policy contention is split resolution (which is supported by e.g. SPIN and VINO). It implies a divisions into global decisions (which are done by the kernel) and local decisions (which are done by the application). For instance the kernel may allocate resources to processes or groups of processes which in their turn make their own allocation decisions. An example of this is the distinction between scheduling of CPU between processes and scheduling between threads inside each process.

3 OBJECT ORIENTATION

The object-oriented paradigm is important in most related work on extensibility in operating systems. The desire for reuse and evolution is e.g. supported by inheritance. Encapsulation is useful for safety. The use of object-oriented frameworks supports the implementation of generic services which can be specialised for specific critical behaviour of for specific hardware. Careful design may then separate out critical parts and make them easily replaceable by subclassing.

3.1 The open implementation approach

We claim that customisability (and extensibility) is closely related to opening up the operating systems abstractions and make certain aspects of implementation controllable and replaceable by applications.

The Open Implementation Approach (see e.g. [Kiczales93a]) is to expose implementation issues in an orderly manner without sacrifying the benefits of abstraction. The reason is the following (according to Kiczales):

> *It isn't possible to hide all implementation issues behind an abstraction barrier because not all of them are "just details", some are instead crucial implementation strategy issues for which the resolution will invariably bias the performance of the resulting implementation.*

A promising approach to this is to use reflection to control non-functional aspects through meta-object interfaces. According to Kiczales, a new approach to controlling complexity will be separation rather than hiding. So, where implementation issues cannot be hidden, they are exposed, but separate from the functional interfaces. The goals of this separation is to let application programmers be able to:

- Control the service implementation when they need to
- Deal with behaviour and implementation largely one at a time.

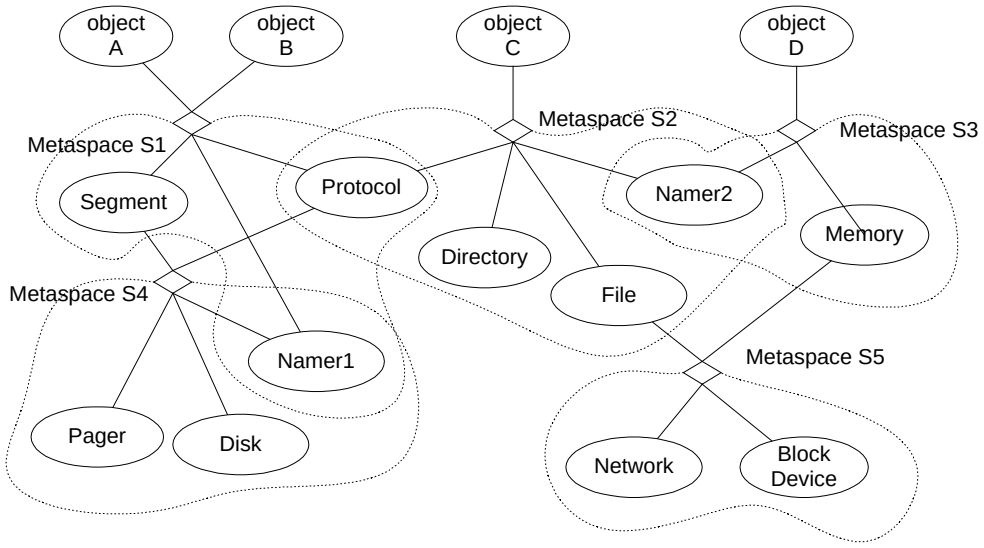
Clients should only be exposed to the meta-interface in exceptional circumstances, e.g. in performance critical sections of their code.

3.1.1 Reflection in operating systems

Operating systems that investigate reflection include Apertos [Yokote92] and μChoices [Li96]. Apertos is based on a model of reflection which is characterised by:

- Strict Object/metaobject separation. An object is a container of information,. The semantics of its behaviour is defined by a set of metaobjects (called a metaspace).
- Metahierarchy. Since a metaobject is an object itself, it has a metaspace.
- Object migration, i.e. objects move between metaspaces. Migration is regarded as a basic mechanism in Apertos. O.S. services may be implemented using migration. For instance a object may be moved to a metaspace that supports persistent storage. The idea of migration between metaspaces also supports heterogeneity.

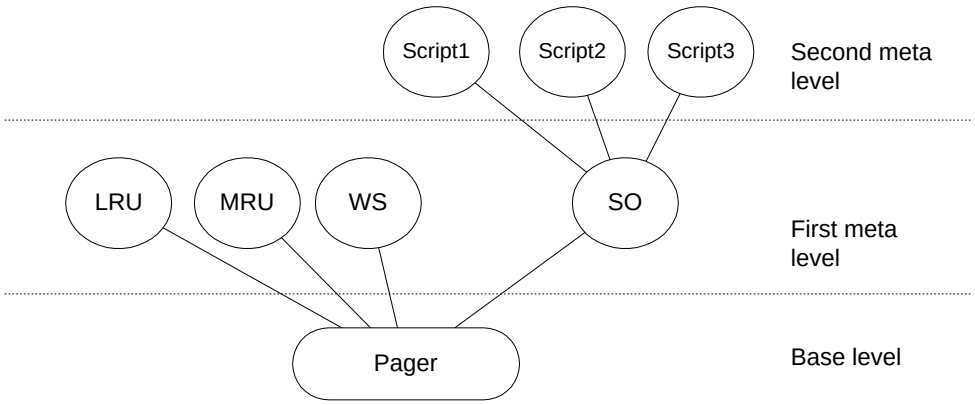
The figure below illustrates the object/metaobject separation and the metahierarchy.



When implementing Apertos, the following is introduced: (1) The MetaCore which is a metaobject which has no metaspace of its own (it is the root of the metahierarchy). It can be regarded as a microkernel and the root of the meta-hierarchy. It is located on each computer and provides common primitives for object execution. (2) Reflectors which represent the metacomputing defined by a metaspace (group of metaobjects).

μChoices adopt some of these ideas and use reflection to construct customisable operating system components like e.g. virtual memory. The architecture defines three levels: The base level, the first and the second meta-level. The base level implements primitive functionality. The first meta level defines the meaning of operations that are included as part of the components behaviour. This level can impose customisable policies by

binding specific operation to specific algorithms (e.g. a choice between LRU, MRU or WS for the paging system). Objects at the first meta level may include special scripting objects. Their semantics are defined by the particular script they execute. Scripts (Java or TCL) are objects at the second meta-level. The figure below illustrates this architecture.



We can claim that μChoices supports both declarative meta-protocols (client may choose between pre-defined implementation choices) and imperative meta-protocols (client can use scripts to implement their own algorithms).

4 EXAMPLE SYSTEMS

In this chapter, we briefly introduce two examples of operating systems that supports customisability and extensibility. We introduce Nemesis which has an Exokernel (or microkernel) like architecture and μChoices which provide dynamic customisability and extensibility by exploiting the idea of reflection and by using interpreted scripts.

4.1 Nemesis

The Nemesis kernel [Roscoe95a] was developed at the University of Cambridge Computer Laboratory. Nemesis is a deliverable of the Pegasus project which is a collaboration between the University of Cambridge, the University of Twente and APM Ltd.

4.1.1 Motivation

The main motivation for developing Nemesis is the need for fine grain resource control, to accommodate quality of service requirements from real-time multimedia applications. One should also be able to dynamically renegotiate resource guarantees given by the O.S. to applications.

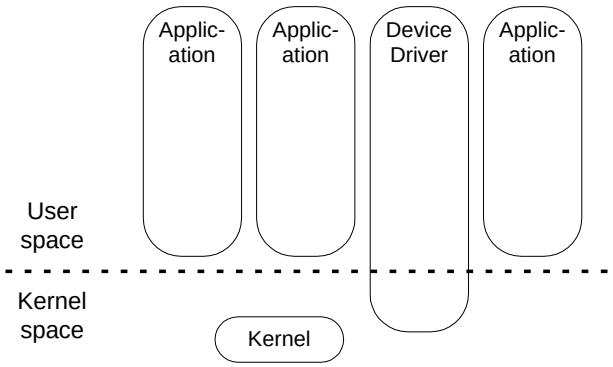
A specific goal is to reduce QoS crosstalk between applications sharing the machine and enable useful resource guarantees to be made. QoS crosstalk can occur when more than one client shares one service, e.g. when more than one stream is multiplexed onto a single lower-lever channel or when more than one client simultaneously use a O.S. server (including the kernel). The load from one client may affect the QoS seen by the other, because of resource contention or concurrency control.

4.1.2 Architecture

To meet the goals mentioned above, the O.S. should account for as much of the time used by an application as possible, without the application losing control over its resource use.

As mentioned in section 2.1, Nemesis takes a Exokernel-like approach. The system is organised as a set of domains (analogous to processes) which are scheduled by a very small kernel. Instead of being located in the kernel or server processes, system services are placed as much as possible in client protection domains and scheduled as part of the client. Communication

between domains should only occur when necessary to enforce protection and concurrency control. Multiplexing of services is then done at as low a level of abstraction as possible. The architecture is illustrated by the figure below.



4.1.3 Main features

We highlight the following main features of Nemesis:

- Structuring tools in the form of modules, objects and typed interfaces within a single address space. These help reduce the complexity of the system from the programmer's viewpoint and enable sharing of text and data between applications.
- A scheduler which delivers useful QoS guarantees to applications in a highly efficient manner. Integrated with the scheduler is an inter-domain communication system which has minimal impact on resource guarantees.
- A framework for high-level inter-domain and inter-machine communication, based on the RPC paradigm. This framework permit QoS negotiation when a communication binding is established. The Nemesis binding model borrows many concepts from the ANSA binding model [Otway94] and supports both implicit and explicit binding.

4.1.4 QoS based scheduling in Nemesis

The Nemesis kernel offers a flexible scheduling service, based on Quality of Service contracts negotiated with the domains to be scheduled. An admission test needs to be carried out and it fails if there isn't enough CPU bandwidth available. Scheduling domains may allocate a QoS (mainly a share of the processors bandwidth), specified by a tuple of four parameters denoting the length of timeslices to be received from the CPU, the time between them, unblocking latency and an indication if the domain is willing to accept extra CPU if available.

Scheduling domains may either be contracted domains (with QoS guarantees) or best-effort classes. The latter may contain one or more domains which are scheduled according to one of many available algorithms (e.g. simple round robin). This approach allows a portion of the CPU

bandwidth to be allocated to domains without special timing requirements, helping to avoid starvation.

4.1.5 Nemesis as an extensible system

The extensibility of Nemesis is supported by important features of object orientation: abstraction and encapsulation, in addition to dynamic loading and linking of code-modules.

Operating system functionality is mostly implemented as small library modules that may be dynamically loaded and linked into application domains. A module is an unit of loadable code analogous to an object file. All code of Nemesis is part of some module. A module implements some object's interfaces and its constructors.

Modules are components which interact only through well-defined interfaces, and they are denoted by interface references (analogous to a pointer to the interface). Interfaces are instances of abstract data types, and they are strongly typed. There is a notion of subtyping, and interface-types are defined in an IDL.

This linkage model supports relatively fine grain extensibility. Modules are typically small and contains the implementation of at least one object type. The principle of abstraction and encapsulation (through strongly typed interfaces) helps programmers write safe O.S. extensions or applications.

Nemesis also offers a name service and dynamic typing, which supports dynamic extensibility. Extensions may even be implemented in the interpretative language CLANGER [Roscoe95b]. By using the run-time naming and typing system, CLANGER allows invocation of interface operations from different modules.

4.2 μChoices

The μChoices operating system [Li96, Tan95] is a research prototype which is developed at the Unversity of Illinois at Urbana-Champaign as a successor of the Choices operating system.

A main motivation is to investigate the construction of an open architecture for reflective computation (see section 3) and dynamic customisation using a flexible scripting language in the operating system kernel.

μChoices adopt a flexible dynamic customisation scheme where user-level policies and mechanisms can be integrated with the system through controlled addition and deletion of scripts. Scripting enable secure embedding of per-application code in the kernel, without needing to rely on a trusted compiler (like in e.g. VINO).

4.2.1 Architecture

The architecture of μChoices is based on a architecture consisting of the following components:

- A nanokernel at the lowest level. A nanokernel encapsulates the hardware and provide basic mechanisms for implementing higher level abstractions such as processes, timers and virtual memory.
- A microkernel interface to the rest of the system. This encapsulates the microkernel data structures and algorithms and separates them from other subsystems of the operating system.
- Operating system servers and API's.

μChoices consist of different O.S. subsystems, realized as separate modules. They are implemented as independent object-oriented frameworks that interact only through well defined interfaces. There is no inheritance between modules. Modules are customised through subclassing of their frameworks[2].

4.2.2 μChoices as an extensible system

As we said above, modules are customised through subclassing and they are compiled and linked into the kernel. This may be regarded as static extensibility.

A main goal of μChoices is to investigate dynamic extensibility, and it does so by allowing applications to download scripted software agents into the kernel, typically for aggregating multiple system calls and thereby minimising kernel/user space boundary crossings. Trusted scripts can provide kernel-level processing of continuous media streams between different devices. Scripts may for instance control system processing of video frames arriving from the network, to the display, without interference or crossing the protection domains.

Scripts are implemented in a scripting language similar to TCL, but using Java is considered as the next step to improve performance. Java is a safe language that is compiled to a intermediate form that is interpreted by a virtual machine which does additional safety checks in run-time. Java then offer both compile-time and run-time protection.

4.2.3 Customisation through a meta-level architecture

As we says in section 3, μChoices exploit the idea of reflection to construct customisable operating system components like e.g. virtual memory.

[2] The nanokernel itself is a module based on frameworks that is subclassed for different hardware platforms.

Changing or activating different components in the meta-space of an object allows customisation of its behaviour.

The architecture defines three levels (see the figure in section 3.2): The base level, the first and the second meta-level. The base level implements abstract or primitive functionality (like e.g. paging). The first meta level defines the meaning of operations that are included as part of the components behaviour. This level can impose customisable policies by binding specific operation to specific algorithms (e.g. a choice between LRU, MRU or WS for the paging system).

Objects at the first meta level may include special scripting objects. Scripting objects are implemented as in-kernel interpreters. Currently μChoices provide both TCL and Java interpreters. The semantics of scripting objects are defined by the particular script they execute. Scripts (written in Java or TCL) are objects at the second meta-level. Thus, the meta-operations at this level include adding, deleting, updating and running of scripts.

5 CONCLUSIVE REMARKS

Object orientation has proven to be useful in construction of customisable operating systems. Object-oriented frameworks lets the designer separate out critical parts of the implementation which may easily be overloaded. The problem how to give applications control over implementation decisions without sacrifying the benefits of abstraction is met by the open implementation approach. Implementation issues and non-functional aspects are exposed as meta-object interfaces where needed, cleanly separated from the functional interfaces.

In this report we have gone through some important issues and options for designing extensible operating systems: First, extensibility may be static (compile time customisation) or dynamic (run-time). Dynamic extensibility may be accommodated by either the microkernel approach, where extensions are implemented in user-space or the downloadable kernel approach, where they may be inserted or replaced in the kernel by applications.

Another issue is protection from erroneous extensions. Protection against illegal memory access is done by hardware (the microkernel approach) or by software. Resource monopolising at the other hand may be tackled by detecting the problem and killing the offending extension.

Other design issues is the persistence of extensions, extension granularity and the resolving of conflicts between extensions.

Scripting[3] seems to be a promising approach to implementing/specifying extensions. [Li96] has demonstrated that scripts (especially when scripts are written in object-oriented languages) fits nicely into the model of reflection (meta-objects may be represented by scripts). Interpreted extensions are attractive alternatives to compiled ones when addressing code portability and safety, but the execution overhead limits their use.

In this context the Java language seems to offer a promising compromise, since it is designed as a safe language that combines compilation and interpretation. It is expected that its execution will be almost as fast as compiled languages as just-in-time compiling of byte-code is incorporated in the Java virtual machine.

[3] A related approach, which we have not examined in this paper is dynamic code generation (see e.g. [Engler96]). Experiences has shown that this has a potential to improve O.S. or application performance significantly.

REFERENCES

[Accetta86]
M. Accetta, R. Baron, W. Bolosky, D. Golub, D. Rashid, A. Tevanian, M. Young, Mach: a New Kernel Foundation for UNIX Development, 1986 Summer USENIX Conference, July 1996

[Bershad95]
B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, Extensibility, Safety and Performance in the SPIN Operating System, Proc. 15th symposium on Operating Systems Principles, December 1995.

[Cheriton94]
D. R. Cheriton, K.J. Duda. A caching model of operating system kernel functionality, Proc. First Symposium on Operating System Design and Implementation. USENIX, November 1994

[Engler95]
D. R. Engler, M. F. Kaashoek, J. O'Toole Jr., Exokernel: An Operating System Architecture for Application-Level Resource Management, Proc. 15th symposium on Operating Systems Principles, December 1995.

[Engler96]
D. R. Engler, W. C. Hsieh, M. F. Kaashoek, " 'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation", To appear in POPL'96.

[Kiczales93]
G. Kiczales, J. Lamping, Operating Systems: Why Object-Oriented?, Proc 1993, International Workshop on Object Orientation in Operating Systems (IWOOOS'93)

[Li96]
Y. Li, S. M. Tan, M. L. Sefika, R.H. Campbell, W. S. Liao, Dynamic Customization in the Choices Operating System, Proc. Reflection'96 conference, San-Fransisco, April 1996

[Montz94]
A. B. Montz, D. Mosberger, S. W. O' Malley, L. L. Peterson, T. A. Proebsting, J. H. Hartman, Scout: A Communications-Oriented Operating System, Dept. of Computer Science, University of Arizona, Technical Report TR 94-20.

[Otway94]

D. Otway, “The ANSA Binding Model”, ANSA Phase III document, APM.1314.01, October 1994.

[Roscoe95a]
T. Roscoe, “The Structure of a Multi-Service Operating System”, Ph. D. Thesis, University of Cambridge, 1995

[Roscoe95b]
T. Roscoe, “CLANGER: An Interpreted Systems Programming Language. ACM Operating Systems Review, 29(2), April 1995.

[Seltzer96]
M. I. Seltzer, Y. Endo, C. Small, K. A. Smith, Issues in Extensible Operating Systems, Pending publication from IEEE Press.
(see URL http://www.eecs.hardward.edu/~vino/vino)

[Small95]
C. Small, M. Seltzer, Structuring the Kernel as a Toolkit of Extensible, Reusable Components”, Proc. 4th International Workshop on Object Orientation in Operating Systems (IWOOOS’95), August 1995, IEEE press.

[Tan95]
S. M. Tan, D. K. Raila, R. H. Campbell, “An Object-Oriented Nano-Kernel for Operating System Hardware Support”, Proc. 4th International Workshop on Object Orientation in Operating Systems (IWOOOS’95), August 1995, IEEE press.

[Yokote92]
Y. Yokote, The Apertos Reflective Operating System: The Concept and its Implementation, Proc. OOPSLA’92, ACM, pp. 414-434, October 1992.