The active object system design and multiprocessor implementation

Author(s):
Muller, Pieter Johannes

Publication Date:
2002

Permanent Link:
https://doi.org/10.3929/ethz-a-004453415

The Active Object System
Design and Multiprocessor Implementation

Diss. ETH No. 14755

The Active Object System Design and Multiprocessor Implementation

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH Zurich)

for the degree of
Doctor of Technical Sciences

presented by
Pieter Johannes Muller
M.Sc., Stellenbosch University
born 11th July 1968
citizen of the Republic of South Africa

accepted on the recommendation of

Prof. Dr. J. Gutknecht, examiner
Prof. Dr. N. Wirth, co-examiner

2002

Bit by bit,
Putting it together ...
Piece by piece—
Only way to make it work.
Every moment makes a contribution,
Every little detail plays a part.
Having just a vision's no solution.
Everything depends on execution:
Putting it together—
That's what counts.

— with apologies to Stephen Sondheim.

Acknowledgements

I am indebted to several people, without whom completing this dissertation would not have been possible.

Prof. Jürg Gutknecht enthusiastically supported my project and gave me free reign to pursue my ideas by providing a comfortable and stimulating working environment. Prof. Niklaus Wirth kindly accepted to be co-examiner and provided much-appreciated encouragement. Together, their work inspired me to pursue my doctoral studies in the first place.

I was fortunate to have three outstanding colleagues working closely with me on the system. Patrik Reali tirelessly maintained the compiler and livened up our office with his jokes. Thomas Frey provided inspiration with his novel graphical user interface and countless applications. Bernhard Egger ported the system to the ARM architecture and greatly improved the installation process.

Prof. Thomas Stricker generously co-sponsored the six-processor machine used during development and testing. Christian Kurmann and Felix Rauch frequently helped with networking and setup issues.

My other Computer Systems Institute friends and colleagues, especially André Fischer, Erich Oswald, Marco Sanvido, and Emil Zeller, ensured a pleasant working environment.

Numerous students and others outside the ETH contributed to the Oberon and Aos systems and motivated us with the trust they placed in our work.

Jaco Geldenhuys and Paul Reed scrutinized the manuscript and provided many corrections and helpful suggestions for improvement. Brian Kirk provided constructive comments on an early version of the design.

Finally, I'd like to thank my wife Eva for her love, tolerance, and inspiration, and my parents for their unending love and support.

Contents

Abstract

Today's pervasive operating systems are large agglomerations of software with growing resource requirements in every new release; both for their development and their operation. Fortunately, with the Oberon project Wirth and Gutknecht have demonstrated that it is possible to develop interesting, practical systems with light resource requirements.

We describe the design and implementation of a generally applicable system more powerful than Oberon, but still comparatively simple and transparent. It is based on a lean and efficient extensible kernel intended for varied hardware environments — servers, workstations, embedded control systems, and small mobile devices.

The kernel primarily acts as a runtime environment for the Active Oberon language, which uses an *active object* concept for concurrency, and *exclusive regions*, combined with a general *await* statement, for synchronization. In a related project the kernel was reused as the basis for a Java virtual machine, demonstrating its flexibility.

The system was implemented for Intel IA-32 symmetric multiprocessor machines, where it schedules active objects to run in parallel, but it also runs on singleprocessor machines. A student has ported it to the Intel StrongARM processor, popular in mobile devices.

This work makes three main contributions. It resulted in the release of a reliable and flexible system that is powerful enough for real applications. It relates experience with active objects and the elegant await synchronization statement. Finally, it serves as a new example of the *lean software* approach to system design. Arguably, by focusing on the essential, such systems can perform their task more securely and reliably than conventional operating systems.

Zusammenfassung

Die heute verbreitetsten Betriebssysteme sind grosse Ansammlungen von Softwareelementen, und bei jeder neuen Version wachsen die Ressourcenanforderungen, sowohl für die Entwicklung als auch für den Betrieb. Wie Wirth und Gutknecht indessen mit dem Oberon-Projekt belegen konnten, ist die Entwicklung interessanter, praktischer, mit wenig Ressourcen auskommender Systeme durchaus möglich.

In der vorliegenden Arbeit werden Design und Implementierung eines allgemein einsetzbaren Systems beschrieben, das leistungsfähiger als Oberon ist, bei vergleichbarer Einfachheit und Transparenz. Das System basiert auf einem schlanken und zugleich leistungsfähigen, erweiterbaren Kern, der sich für verschiedenste Hardwareumgebungen wie Server, Workstations, eingebettete Kontrollsysteme und kleine Mobilgeräte eignet.

Der Kern dient hauptsächlich als Laufzeitumgebung für die Active-Oberon-Sprache, welche die Konzepte *aktive Objekte* für Nebenläufigkeit und *exklusive Regionen*, kombiniert mit einem allgemeinen *await*-Statement, für Synchronisierung verwendet. Die Flexibilität des Kerns wurde in einem verwandten Dissertationsprojekt aufgezeigt, bei dem er als Grundlage für eine Java virtuelle Maschine verwendet wurde.

Das System wurde für Intel IA-32 symmetrische Multiprozessor-Rechner implementiert, wo es aktive Objekte parallel ablaufen lässt, funktioniert aber genauso auf Einprozessor-Maschinen. In einem Studentenprojekt wurde es auf den Intel StrongARM Prozessor portiert, der häufig bei Mobilgeräten eingesetzt wird.

Folgende Hauptbeiträge werden in der vorliegenden Studie präsentiert: Ein zuverlässiges und flexibles Betriebssystem wurde entwickelt, das leistungsfähig genug ist für echte Anwendungen. Im weiteren wird über die gemachten Erfahrungen mit aktiven Objekten und dem eleganten *await*-Synchronisierungsstatement berichtet. Schliesslich wird ein neuartiges Beispiel des *lean software* Ansatzes zum Systemdesign aufgezeigt. Derart entwickelte Systeme dürften, indem sie sich auf das Wesentliche konzentrieren, ihre Aufgabe sicherer und zuverlässiger erfüllen als herkömmliche Betriebssysteme.

Chapter 1

Introduction

1.1 Motivation

Modern pervasive operating systems are large agglomerations of software built in years of work by large groups of developers. Their resource requirements seem to mirror the effort placed into constructing them. Fortunately, the Oberon project has demonstrated that it is possible to develop interesting, practical systems with little manpower [134]. The outcome of the Oberon project was a simple, efficient and transparent operating system that was later described as *lean software* [38, 79, 133] — software that is concentrated on the essential and excludes the inessential.

In this dissertation we attempt to follow in the footsteps of the Oberon project to design and implement a multiprocessor operating system based on a lean kernel for active object-based systems. The aim was to develop a generally applicable system more powerful than Oberon, but still comparatively simple and transparent.

Our motivation is the belief that system software does not have to be gigantic to be useful or powerful. A *sustainable* approach to computing is more desirable than a reliance on Moore's law to support ever-growing software systems, sometimes with countless 'features' of doubtful utility (but great marketability).

The specific goals for our project were to:

1. Design a lean and efficient kernel for general active object-based systems. The kernel should be applicable in many areas, e.g.,

in an operating system similar to Oberon, in a server operating system, in an embedded control system or as an operating system for small devices (i.e., mobile computers). Therefore, it should also be realizable on a wide range of hardware, from powerful multiprocessor server machines down to modest hand-held and wearable devices.

2. Implement the kernel on powerful industry-standard machines, specifically the Intel 32-bit multiprocessor system architecture.

3. Use the kernel as a testbed to implement a general-purpose active object-based operating system. The system should host its own development environment and other applications like servers.

4. Port the current ETH Oberon system to run in an active object on the system, providing compatibility with existing Oberon applications.

To achieve the flexibility demanded by the first goal, while keeping to the standard of leanness and transparency set by Oberon, the kernel should be little more than a runtime environment for a type-safe active object-based language, with the possibility of 'plugging in' additional capabilities.

The *active object runtime system* (Aos kernel) that was developed to fulfil these goals forms the basis of an extensible operating system (Aos system), which was created by implementing several system services using active objects.

1.2 Background

Oberon language. An important reason for the Oberon system's leanness is that it is wholly based on the equally lean and powerful Oberon language [98, 132], which allows the system to be structured as a collection of shared dynamically-loaded modules. We adopted a similar language-based approach from the start.

Active Objects. Already before the start of our project Gutknecht started to extend the Oberon language with direct support for concurrency and *active objects* were identified as a suitable abstraction [45].

The resulting Active Oberon language was selected as the basis for our system as it retains the leanness and transparency of Oberon and has a powerful and clean concurrency model.

One may argue that the Oberon system's single-process multitasking model contributes to its leanness, because it obviates the need for concurrency protection mechanisms. However, this model, although well-suited to an isolated single-user system, is insufficient in a more general environment including networking or other inherently concurrent components. Additionally, it is incompatible with multiprocessor machines, and we therefore had to replace it with a more general concurrency model, at the cost of adding some complexity to the design.

Native Oberon. The original Oberon system (called *Ceres Oberon* in this document) was implemented natively on the custom-designed Ceres workstation with little concern for portability. Nevertheless, due to its leanness, modular system design and Oberon language abstraction, it was soon ported to run as an application on many different operating systems and hardware architectures. The system was also extended to support persistent objects [44] and a document-oriented user interface and component architecture [63].

As there was a need for an Oberon system running natively on modern personal computers, the author ported the ETH Oberon system to the industry standard PC architecture in a previous project. The resulting *Native Oberon* system [74] resembles the original system for Ceres, but due to the more diverse and complex hardware environment some lower-level modules were expanded and generalized. By concentrating on open hardware standards like IDE, VGA, VESA and PCI the implementation effort was kept within bounds while supporting a wide range of hardware.

Native Oberon has proven itself in teaching and day-to-day use and has been the basis of many projects: undergraduate [24, 35, 41, 58, 59, 87, 115], diploma [25, 57, 73, 111], postgraduate [30, 77] and industrial [49, 110]. A notable project was its adaption into a prototype active object runtime system and singleprocessor server system called Eamon [30, 31].

Although Native Oberon is a separate system, it has influenced the Aos kernel and system significantly, and the experience gained during its development is referred to in some of the discussion here. The system was initially used as cross-development environment for Aos. Finally, it was ported to run as an application on Aos, allowing existing ETH Oberon applications to run on Aos, thereby allowing the system to host its own development environment.

1.3 Contributions

The main contributions of this work are the following:

1. It resulted in the release of a reliable and flexible system that is powerful enough to be used in real applications.

2. It relates experience with active objects in the Active Oberon language. Although active objects are little more than objects combined with lightweight processes, a controversial aspect of the language is its general *await* statement for process synchronization. We show with practical experience that this elegant synchronization concept can be applied effectively in a real system.

3. It serves as a new example of the *lean software* approach to system design. For example, a system with only 50KB of kernel code and 150KB of operating system services and application code can perform useful tasks, such as serving dynamic web pages on an off-the-shelf multiprocessor machine. Arguably, by focusing on the essential, it can perform its task more securely and reliably than a conventional operating system.

1.4 Overview

Chapter 2 describes the active object computing model and the Active Oberon language.

Chapter 3 describes the structure of the active object system (Aos system).

Chapter 4 describes the design of the active object runtime system (Aos kernel).

Chapter 5 describes the multiprocessor runtime implementation in detail.

Chapter 6 describes the active object-based device and service abstractions provided by the Aos system.

Chapter 7 presents various application case studies.

Chapter 8 compares the system with related systems and presents performance measurements.

Chapter 9 concludes and discusses future directions.

Appendix A shows the sizes of the modules of the implementation.

Appendix B contains additional technical notes on the implementation.

Appendix C describes an alternative interrupt handling model.

Chapter 2

Active Object Concepts

This chapter summarizes the active object computing model and the Active Oberon language.

2.1 Active Object Computing Model

When adding concurrent processes to an object-oriented system, one of two viewpoints can be taken, illustrated in figure 2.1. On the left is a system of objects with *external processes* operating on them. On the right is a system of *active objects* with internal processes specifying their intrinsic activity.

In the external process model processes and objects are independent entities. Processes modify the state of objects by calling their methods. They interact with each other by operating on shared objects. For example, in the figure, process P and Q interact by accessing shared object Y, which must be suitably protected against conflicting accesses with some synchronization mechanism.

In the active object model, a process is encapsulated in an object. A process is always an integral part of an object, created when the object is instantiated. As in the first model, the active objects can interact by calling the methods of shared active or non-active objects. The differences between the two viewpoints may be subtle and controversial [66], but we believe active objects present a more natural, integrated model of concurrency.

Similar to normal objects, active objects have state and supply meth-

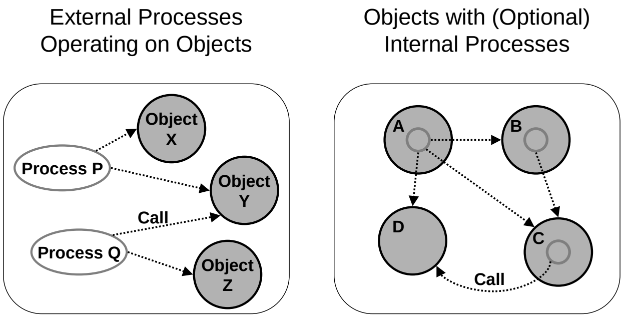


Figure 2.1: Objects and processes: two viewpoints.

ods which can be called to access the services provided by the object. An object's methods can access and modify its state in a controlled way. As active objects exist in a concurrent environment, a mechanism for coordinating concurrent access to their state by their methods must be provided.

To coordinate conflicting concurrent accesses to object state the programmer specifies *exclusive regions* in the program text of an object. These are critical sections of the program that manipulate the state of an object intended to be shared between processes. The runtime system guarantees that at most one process will be active in an exclusive region, thereby avoiding conflicting state manipulations.

Concurrent actions are synchronized using an *await statement* to specify an arbitrary boolean condition as precondition for continued program execution in an object. If the condition is false, the program is suspended until the condition is established by an active object calling a method of the first object. In this way method calls act as communication mechanism between active objects.

Figure 2.2 shows a population of communicating objects over time and illustrates some active object synchronization situations. Active object B starts by calling a method of non-active object R and is suspended inside R, waiting for condition p. Later, active object A calls a method of R and establishes the condition. The method call by A returns and also enables the earlier method call by B to return, as p is now true. Now A and B continue running independently. A performs another method call of R, and B calls a method of active object C, where it is suspended waiting for condition q. In this case, the process of C itself establishes the condition after some time has passed, enabling B to continue running (C could be a timer object providing a delay service).

The exclusive region concept is similar to the *critical region* of Brinch Hansen [18]. An exclusive region that encompasses the whole body of a method corresponds to Brinch Hansen and Hoare's *monitor procedure* [19, 47]. Unlike monitor procedures, active object methods are not mandatorily exclusive.

The await statement was first described by Brinch Hansen in connection with critical regions and his *shared classes* monitor concept [18, 19]. However, most implementations of monitors use Hoare's *condition variables* [47] (based on Brinch Hansen's *event queues* [18]) as synchroniza-

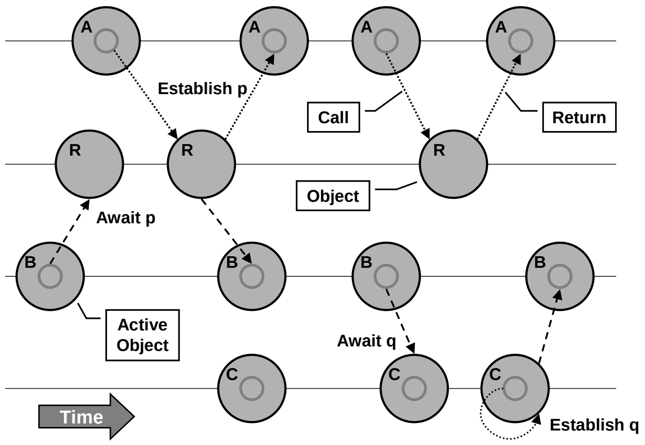


Figure 2.2: Example population of communicating objects.

tion mechanism instead. The await statement is more elegant than condition variables, but was perceived to be "too inefficient for general use in operating systems" [47, Conclusion]. With our system we demonstrate that this is not necessarily a problem in practice.

A central idea of the monitor (and active object) concept is that the programmer associates an invariant with the monitor object, which holds whenever it is at rest [28]. The invariant is used to characterize the behaviour of the object and make correctness proofs. The appropriateness of monitors for rigorously structuring operating systems has been demonstrated by various projects [61, 62]. When the monitor concept is combined with a module concept, it forms a powerful mechanism for structuring operating systems [56, 129, 130].

It should be noted that the notion of an active object described here is closely related to the *sequential process* model of concurrency [20, 48] rather than the *actors* model. The latter uses data-flow, asynchronous communication, implicit pipelining and no explicit store [5]. Although it is an interesting theoretical framework, it remains to be seen if practical systems and machine architectures can be constructed using this approach. In typical actor example programs [6] large numbers of actors are created, perhaps comparable to the number of recursion levels in sequential solutions of the same problems.

Active objects seem well-suited for programming *software agents*, which are 'intelligent' programs that can autonomously collaborate, adapt and learn to solve problems in a goal-directed way [17].

2.2 Active Oberon Language

The Active Oberon language developed in the group of Prof. Gutknecht at the ETH Zurich [45] is an extension of the Oberon language [132] that supports active objects directly. This section briefly describes the language extensions and presents some examples of their use. The first example is a complete *bounded buffer* implementation, shown in figure 2.3.

2.2.1 Object Types and Methods

The *object types* (classes) of Active Oberon are similar to Oberon's record pointer types, but can have *methods* and *bodies.* An object type

```
MODULE BoundedBuffers;
TYPE
  Item* = OBJECT; (* generic object *)
  Buffer* = OBJECT
    VAR h, n: INTEGER; B: ARRAY * OF Item;

    PROCEDURE Get*(): Item;
    VAR x: Item;
    BEGIN {EXCLUSIVE}
      AWAIT(n # 0); (* buffer not empty *)
      x := B[h]; h := (h+1) MOD LEN(B); DEC(n);
      RETURN x
    END Get;

    PROCEDURE Put*(x: Item);
    BEGIN {EXCLUSIVE}
      AWAIT(n # LEN(B)); (* buffer not full *)
      B[(h+n) MOD LEN(B)] := x; INC(n)
    END Put;

    PROCEDURE &Init(max: INTEGER);
    BEGIN (* initializer *)
      NEW(B, max); h := 0; n := 0
    END Init;
  END Buffer;
END BoundedBuffers.
```

Figure 2.3: A generic bounded buffer in Active Oberon.

(e.g., Buffer in figure 2.3) is declared like a record type. Variables of this type, called *object variables*, are always references to *object instances* that are dynamically created with the standard procedure NEW. We use the term *object* interchangeably to refer to object types, object variables and object instances, as it is clear from the context what is meant.

Methods are declared as procedures in the syntactic scope of an object type (e.g., Get, Put and Init in figure 2.3). Methods are intended to provide controlled access to the local data of an object. In an extended object type, methods of the base type can be overridden. Supercalls in an overridden method are marked with the '^' character.

One method per object type can be marked with a '&' character, specifying that it is an *initializer* (e.g., Init in figure 2.3). This method is called automatically when an object is allocated with NEW. The second and subsequent actual parameters of NEW are matched to the formal parameters of the initializer. They are normally used to provide initialization parameters to an object, but VAR parameters can also be used to return information from the initializer. The initializer is a normal method that can be called, overridden and supercalled like any other. If the object type is exported, the initializer is automatically exported.

A comma-separated list of modifier keywords parenthesized by '{' and '}' can be written after the BEGIN of any statement block, and specify some modification of the default semantics of the block. The most important modifiers are described below.

The EXCLUSIVE modifier on a statement block defines the block as a *critical section* of the immediately enclosing object (or module). The compiler and runtime system ensure that the exclusive regions of an instance are entered by at most one process at a time. For example, the Get and Put methods in figure 2.3 are declared exclusive, as they modify the state of the buffer.

Typically, the programmer associates an invariant with an object type. Initializers and exclusive methods are the tools provided by the language for maintaining object invariants, and hiding internal object states. The initializer of an object establishes its invariant, and the exclusive methods maintain it. If a method is not declared exclusive, it is possible for it to observe inconsistent object states.

2.2.2 Synchronization

Synchronization between processes is performed by the AWAIT statement, which takes an arbitrary boolean expression B as parameter. B is a *condition* for continued execution, which means AWAIT only terminates when B has become true. Otherwise, the current process is delayed until B is found to be true by the runtime system (e.g., the AWAIT in Get in figure 2.3 waits until the buffer is non-empty). To ensure well-behaved synchronization, an await statement must be enclosed in an exclusive region. When the await statement delays execution due to an unsatisfied condition, the exclusive region is temporarily released to other processes, until the condition is satisfied. At that time the waiting process has to re-enter the exclusive region. Therefore, the precondition of the AWAIT statement is usually the same as the invariant $\mathcal{I}$ of the exclusive region, and the postcondition is $\mathcal{I} \wedge B$.

2.2.3 Active Objects

An object type declaration can have a body, which is an anonymous statement block situated at the end of the declaration, similar to a module body. The body is invoked at runtime after the initializer of the object (if any) has terminated successfully. It is usually used to specify the intrinsic behaviour of an active object, in combination with the ACTIVE modifier below.

The ACTIVE modifier on an object (or module) body specifies that a new process is created to execute the body. This is used to generate an active object. When the process reaches the end of the body it terminates normally. Object bodies may also be EXCLUSIVE.

Like methods, object bodies can be overridden in extensions of the object type. All the bodies of an extended object will be invoked sequentially when an instance is created, starting with the body of the base type. In the case of multiple ACTIVE bodies, multiple processes will be created. This could be useful to model a class with different concurrent activities at different levels of the type extension hierarchy.

Figure 2.4 shows an example of an active object in a networking application. The AwaitResponse procedure reads a message from the network connection and reacts on it.

2.2.4 Miscellaneous

Anonymous object types may also be declared directly in variable declarations, similar to anonymous record types. For example, in figure 2.5, the producer variable has an anonymous object type. The object instance still has to be allocated with NEW, because the producer variable only stores a reference to the instance. The example also shows how a non-active object (Buffer from figure 2.3) can be extended with an active body.

If a process causes an exception, it is terminated, unless the SAFE modifier is specified on the active body that created it. In that case,

```
Receiver = OBJECT
  VAR c: Connection;

  PROCEDURE &Init(c: Connection);
  BEGIN SELF.c := c
  END Init;

BEGIN {ACTIVE}
  REPEAT AwaitResponse(c) UNTIL c.res # Ok
END Receiver;
```

Figure 2.4: An active object that awaits and reacts to messages on a network connection.

the process's stack is reset, and it restarts executing at the start of the body.

In the syntactic scope of an object type, the keyword SELF can be used as a reference to the object instance (e.g., inside Init in figure 2.4).

The built-in type OBJECT is the implied base type of all objects. A variable of this type can store a reference to any object instance. It is usually used in combination with type tests and type guards, to implement generic object operations.

When an active object is instantiated, exactly one process is created for each of its ACTIVE bodies. Every process is associated with exactly one active object. The standard function procedure ACTIVE() returns a generic reference to the active object associated with the current executing process. This facility can be used by resource objects to obtain a reference to their client active objects.

A BEGIN-END block can be used anywhere a normal statement is allowed. The EXCLUSIVE modifier may also be used on a block statement. This allows more fine-grained specification of critical sections.

Procedure types have been extended so that *procedure variables* can also store references to methods, not just to normal procedures. A procedure variable now also stores a reference to an object instance, which is NIL in the case of normal procedure variables. The type of a method is identical to that of a procedure with the same signature, making it compatible with procedure variables of the same procedure type.

```
TYPE
  Item = OBJECT ... END Item;

VAR
  producer: OBJECT (BoundedBuffers.Buffer)
    VAR i: Item; "state variables"
  BEGIN {ACTIVE}
    "initialize state"
    LOOP
      "compute item i and modify state"
      Put(i)
    END
  END;

VAR
  consumer: OBJECT
    VAR i: OBJECT;
  BEGIN {ACTIVE}
    LOOP i := producer.Get(); "use item i" END
  END;

BEGIN
  NEW(producer, BufSize); NEW(consumer)
END.
```

Figure 2.5: Extending a bounded buffer to form an active producer object.

```
MODULE TestBuffer;
IMPORT BoundedBuffers, In, Out;

TYPE Item = OBJECT VAR n: INTEGER END Item;
VAR b: BoundedBuffers.Buffer;

PROCEDURE Put*;
VAR n: INTEGER; i: Item;
BEGIN
  In.Open; In.Int(n);
  IF In.Done THEN NEW(i); i.n := n; b.Put(i) END
END Put;

PROCEDURE Get*;
VAR i: OBJECT;
BEGIN
  i := b.Get(); Out.Int(i(Item).n, 1); Out.Ln
END Get;

BEGIN
  NEW(b, 10)
END TestBuffer.
```

Figure 2.6: A test module for the bounded buffer of figure 2.3.

Like the Oberon language, Active Oberon is also intended for system programming, so some low-level facilities are provided. The SYSTEM pseudo-module allows direct access to memory, registers and I/O ports, and has a typecasting operator for circumventing the strong type system. This module is implemented directly by the compiler and its procedures are translated to inline code. When SYSTEM is imported, procedures can also be written in assembler language, using a CODE-END block. Such procedures are assembled directly by the compiler and when marked with the '-' character, they are expanded inline at the call location.

To complete the examples and conclude this section, figure 2.6 shows a test program using the bounded buffer of figure 2.3.

Chapter 3

System Structuring Concepts

This chapter provides an overview of the system structuring concepts used in the active object system (Aos).

3.1 Extensible Systems

Aos is an *extensible* (or *open*) system, which means that there is no strong division between user programs (applications) and the system. A safe strongly-typed language with run-time checks is used to program the system and applications, conserving the integrity of the whole. The system is an open collection of modules, with no inherent difference between system modules and application modules.

In contrast, a *closed* system separates applications and the operating system by assigning each its own independent address space. The closed approach allows system integrity to be maintained in the face of applications that corrupt the memory assigned to them. This is essential when applications are written in unsafe languages, which are less helpful in avoiding and containing programming errors. Examples of closed systems are Unix, Windows NT and microkernel-based systems like Mach.

In an extensible system, applications can call system-supplied operations directly and data can be shared directly between the system and applications, and between different applications, because a single address space is used. In a closed system, applications communicate with the operating system using *supervisor calls* or some other cross-address-space calling mechanism. They communicate with each other indirectly via the operating system, using interprocess communication facilities that usually require serialization of parameters, as different address spaces are used.

Past experience with extensible operating systems developed at the ETH Zurich Computer Systems Institute has demonstrated that this approach can be used to build highly transparent, lean and reliable single-user workstation operating systems (e.g., Medos-2, Vamos, Oberon, ETHOS).

Aos extends the family of ETH-style extensible operating systems with the active object concurrency concept implemented for multiprocessors. Furthermore, its application area is defined much broader than single-user workstation systems, to include server systems, embedded control systems and small hand-held or wearable devices.

Like Oberon, Aos is intended for essentially *cooperative environments*. This does not contradict using it for server applications, even though a server application typically has multiple clients connecting to it on behalf of multiple users. The essential point is that the whole server environment is under control of a single authority, and the network provides a well-defined interface to the multiple clients. If required, a server application can be programmed to share its resources fairly between clients.

3.2 Modular Structure

A common theme of the extensible systems mentioned above is their *modular structure*, based on the facilities provided by a *modular programming language* (in this case, Modula-2 and Oberon). Aos has a modular design inspired by these systems, especially Oberon. According to Parnas [79], 'software jewels' like Oberon owe their elegance to good decomposition principles, good hierarchical structures and good interface design.

An important advantage of the Oberon (and Active Oberon) language is that the import relationships between modules are declared explicitly. The resulting import graph is constrained to be acyclic, which gives the system a *hierarchical structure*. This static framework simplifies the identification of independent subsystems and clarifies the system structure.

While organizing the system into modules, we attempted to follow Parnas's modular design principles [80, 81]. Modules should separate concerns, with each module responsible for a single well-defined function. Dependencies between modules should be minimized, to keep complexity down, and when dependencies exist, they should be made explicit. Modules should be used to hide implementation details and create abstractions. When implementation details are encapsulated cleanly, module invariants can be guaranteed by the implementation, improving the robustness of the system.

Finding a clean modular decomposition is a difficult problem, and in most cases the solution is a compromise between clean abstraction, reduced complexity and efficiency, as noted by Szyperski [118]. He gives some hints for module design, e.g., when two modules are interdependent and would therefore need to import each other, they should better be merged into one larger module. If the resulting module would be too large, a third module containing the shared parts could be factored out of the two modules. This module would most likely expose some implementation details in its interface, and would therefore have to be declared as a private implementation module of the system.

3.3 Module Layers

The modules of Aos are separated into three conceptual layers, as shown in figure 3.1. The upper layer contains Aos and Oberon application modules, and the other two layers form the Aos system proper.

The bottom layer is the *active object runtime system*, which contains the runtime support for the Active Oberon language. This is the only layer that has to be present in all configurations of the system, and it is known as the *Aos kernel*. Its design and implementation are the subjects of chapters 4 and 5.

The middle layer contains *shared system services*, e.g., the communication and file subsystems. Internally, this layer is horizontally separated into modules providing abstract service objects and their concrete implementations. It is separated vertically into collections of modules concerned with independent services. Chapter 6 describes this layer in detail.

Application modules form the top layer of the modular structure. These can either be pure Aos applications, or Oberon applications that

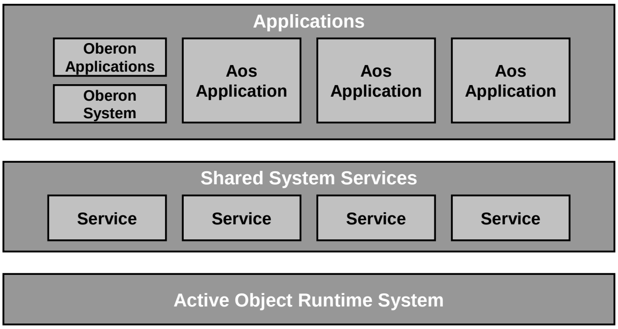


Figure 3.1: Overview of the Aos system structure.

make use of the Oberon for Aos subsystem, which is an emulation of the Oberon system that allows most Native Oberon applications to be compiled and run without change on Aos (cf. 7.1).

The system modules are layered on an abstraction level, but not on a functional level. This means that a lower layer will never rely on abstractions supplied in a higher layer, but could rely indirectly on functionality supplied there. For example, the runtime system provides an abstract module loader interface, which can be implemented in the services layer, with the help of a file system and disk driver located there. In this way the core of the system is kept small and flexible.

As is common in extensible systems the layering is not strict (in the sense that a higher layer may only rely on abstractions supplied by the immediately lower layer), and therefore higher layers may use or extend abstractions supplied on any lower layer. For example, application modules may import and use public modules in the runtime system directly, without having to go through modules of the services layer. Similarly, a device driver can communicate directly with its hardware device using low-level language facilities. By keeping the system layers relatively flat and open, the 'multiple layers of inefficiency' syndrome of strictly layered systems is avoided. Although buffering and batching of operations can improve throughput in such systems, the latency of operations are always adversely affected by the many layers they must pass through.

The whole module hierarchy is *open-ended*, i.e., system services and application programs are simply modules that extend the system's module hierarchy.

3.4 Dynamic Loading

In Aos, as in the Oberon system, there is no concept of a pre-linked executable program. Instead, all modules are *dynamically loaded* into memory when they are first used.

As a module typically depends on functionality exported from other modules, all required lower-level modules are loaded automatically when a higher-level module is loaded. If a required module is already present in memory, the existing instance is reused. Thus module instances are automatically shared between the modules depending on them. This makes a significant contribution to the low storage requirements of the system, as each module is only present once in memory and in the file system.

Modules can be unloaded from memory when no longer required. While a module remains loaded in memory, its global variables represent its state, which can be used for communication between different procedure invocations and processes. Therefore, module unloading is not performed automatically by the system, but can be done under program control, or manually by the user.

The ability to dynamically load modules is indispensable for a flexible extensible system, as it allows extensions to the system to be loaded at runtime.

3.5 Service Modules

A logical approach to design lean and flexible systems is to use a minimal system core or *kernel* that can serve as the basis for the additional components necessary to solve some specific problem [78].

An example of this is the *microkernel* design approach popularized in the nineties, where a small system core provides only very basic support facilities like processes and interprocess communication (IPC), and other operating system functionality is provided by application-level servers that can be added and removed like normal user applications [119]. This design also makes it possible to develop and test interchangeable system components independently.

In microkernel systems, the servers are implemented as processes in separate address spaces, and their clients communicate with them via IPC facilities, e.g., synchronous message passing or remote procedure calls. In early microkernels, this overhead led to performance problems, which were mitigated by later designs [46]. However, even with low-latency and high-bandwidth IPC, the overhead of marshalling parameters into serial form for cross-address-space communication remains.

In response to this, newer monolithic kernel designs such as Solaris [64] and Linux allow 'modules' to be loaded dynamically into the kernel. Although this solves the performance problems associated with application-level servers, it does not provide all the advantages. Loadable modules execute in a specialized environment and can not be developed and tested as normal user-level applications. Additionally, modules are not cleanly supported in the languages used in these kernels.

In Aos, the active object runtime system is used as a small system core, and additional system functionality is added by *service modules* in the system services layer. This mirrors the basic idea of the microkernel design, but avoids the overhead. Service modules are accessed by their clients via normal procedure and method calls. Compared to microkernels and monolithic kernels, this lightweight approach promises much better performance and reduced complexity.

Although service modules conceptually belong to a separate system layer, they are completely normal Active Oberon modules that export their server functionality for clients to import and use. As dynamic loading is used, the service modules required by an application are automatically loaded and started when it is loaded.

3.6 Plugin Modules and Objects

Plugin modules are used in Aos to further improve the flexibility and leanness of the system. A plugin module is used in a situation where a standard system interface is identified that can be implemented in different ways.

A typical example are device drivers for a specific class of devices. In this case an abstract interface is defined for the class of devices, and several different plugin driver modules are implemented for specific members of the class. The abstract device interface is defined in a module of its own and clients that use the device import this module. In a given system configuration the appropriate driver modules are 'plugged in' dynamically into the interface module. In this way the service module and the device modules are indirectly coupled via the interface module (see figure 3.2).

A plugin module is usually responsible for several instances of the object class it manages and therefore instantiates one or more *plugin objects*. For example, a device driver plugin will instantiate one object for every instance of the device present in the system.

Unlike normal modules, plugin modules are not imported directly by their clients. Instead, a client imports the interface module that defines the interface implemented by the various plugin modules. A plugin module also imports this interface module and registers itself there.

For this purpose, a *plugin registry object* is defined by the system. Each interface module exports an instance of this object, which is a collection of plugin objects and serves as a registry where plugin objects

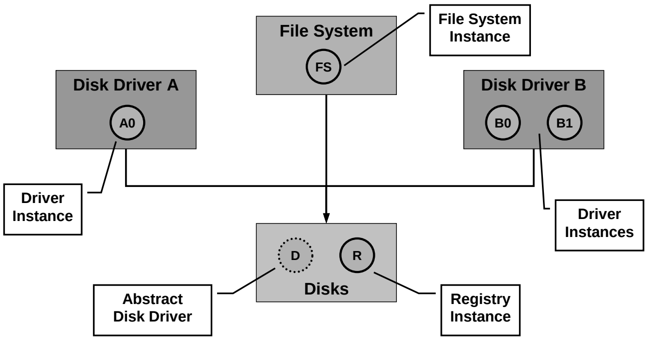


Figure 3.2: Plugin object registry example: disk drivers.

that implement the specific interface are registered by plugin modules, and retrieved by client modules.

Figure 3.2 is an example of how the registry service is used. The four shared rectangles represent modules. The Disks module is the interface module for disk drivers, which defines an abstract disk driver object. This object is extended in concrete disk driver implementation modules A and B, which implement drivers for two classes of disk devices. A file system implementation module imports the Disks module and accesses a disk indirectly using the interface defined by the abstract disk driver object. It does not import any of the disk driver modules directly.

The shaded circles in the figure represent object instances. The Disks module exports a single global variable instance of a plugin registry object (R), which serves as a collection of all disk driver plugin objects. Each disk driver implementation module (A and B) implements a disk driver as an extension of the abstract disk driver object (D). One instance of such an object is created for every physical disk ($A0$, $B0$, $B1$) and registered with R. A file system object finds the driver object for the disk containing its data from the registry.

3.7 Commands

Aos uses a more general implementation of the Oberon system's *command* concept. Commands are indirectly related to system structuring as they provide a mechanism for loose coupling of modules. For example, plugin modules are normally instantiated via commands, while some plugin objects, e.g., file system implementations, are created by generator commands.

The Oberon system's commands are simply *exported parameterless procedures* that are dynamically callable using the module loader. They are used mainly for two purposes: *initiating actions* in the Oberon user interface and *generating object instances.*

The Oberon user interface uses commands as the principle means for the user to initiate actions. The text viewer system translates a mouse click on a text string of the form $M.P$ into a call of command procedure P in module M. If the module is not yet present in memory, it is loaded. Once a module is in memory, its global variables can be used for communication between subsequent command invocations. For example, the display space data structure containing all open viewers (windows) is rooted in a global variable of the Viewers module. Commands operate on this global data which, together with data structures on the heap, represents the state of the user interface.

In the persistent object kernel of Oberon, a *generator* is a command that creates an instance of a specific *object type.* Since command procedures can be called dynamically using the module loader, they allow a client module to call procedures without statically importing their defining module. In fact, the name of the command could be generated at runtime and the command need not even exist at the time the client module is compiled. Thus, generators provide a way to generate object instances from modules that are not directly imported. This is essential when creating modules that manipulate heterogeneous extensible data structures, e.g., the rich text system of Oberon.

In both these uses of commands in Oberon, global variables are used for communication between the invoker of a command and the command itself. In the case of a command initiated by a mouse click, the context of the *M.P* text string is stored in a well-known global variable, where *P* can fetch it to scan its parameters. In the case of generator commands, the generated object instance is assigned to another well-known global variable, where it is fetched by its consumer.

The Oberon command concept can not be applied directly in a multiprocessing system like Aos. The global variables used for communication between a command and its invoker are shared resources that have to be protected from concurrent access by different processes.

In Aos, the notion of a command is generalized so that it is no longer a parameterless procedure. Parameters can now be passed to commands directly and they can return results directly, without resorting to global variables or an explicit locking protocol. A *parameterized command* in Aos is defined as a procedure with a generic pointer parameter and a generic pointer return value. This requires only a small change to the compiler, the module loader and boot linker.

With this simple change, commands turn into a powerful mechanism for realizing independent modules that can work together nonetheless. They are used to instantiate plugin modules and to generate plugin objects without directly importing them, an essential feature in a lean extensible system. They are also used by a user to initiate actions, and unlike Oberon commands, they work reliably even if actions are initiated in parallel, e.g., by a batch command processing facility to run commands in the background using active objects.

Chapter 4

Active Object Runtime Design

In this chapter the design decisions that were made in the development of the active object runtime system (Aos kernel) are discussed. The next chapter describes the details of the runtime system modules.

4.1 Runtime Requirements

Programs in compiled languages are translated by a compiler into machine code that can be executed directly by a machine. Many constructs in these languages, such as assignments, loop statements and expressions, can be translated directly into machine code, but some more complicated constructs, such as dynamic object allocation, can not be completely translated and require the assistance of a *runtime system*, which consists of subroutines and data used by the translated code to perform its functions. Some languages, e.g., Modula-2 and C, require almost no runtime support, and other languages, e.g., Active Oberon and Java, have high-level constructs that require more runtime support.

The primary design requirement for the Aos kernel is to be a complete runtime environment for the Active Oberon language (cf. 2.2). The kernel is tuned for this language, even though it is it is capable of supporting other comparable languages, e.g., Java (cf. 8.3.2). Mechanisms required by other language environments, but not by Active Oberon, can be provided in the form of plugins, to avoid encumbering the kernel unnecessarily.

A runtime system is usually based on the facilities provided by the

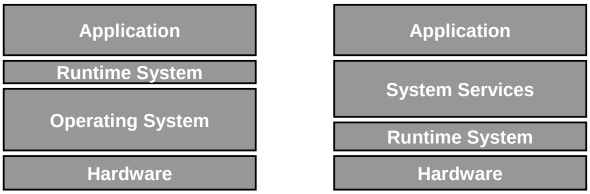


Figure 4.1: Runtime system location (traditional left, Aos right).

underlying operating system, but in the case of Aos the runtime system resides on the bare machine (see figure 4.1). Therefore, two secondary requirements are that it must implement the required runtime support natively in terms of the underlying hardware, and it must support the programming of operating system services. Most operating system services like file systems and communication can be programmed directly in Active Oberon, but device drivers require some interface to the underlying hardware for I/O operations and interrupt handling.

The next section describes the active object language support and section 4.3 describes the facilities for programming native operating system services.

4.2 Active Object Language Support

The Aos kernel is mainly intended as a runtime system for Active Oberon and similar languages. The facilities it supplies for this purpose are:

- Modules and dynamic loading.
- A simple object model with type extension (single-inheritance subclassing) and type-bound procedures (methods).
- Object memory management using garbage collection.
- Lightweight processes, locks and general process synchronization.

4.2.1 Modules and Dynamic Loading

Dynamic loading of modules plays an important role in an extensible system (cf. 3.4). To execute a module, its source code has to be transformed to machine code in memory. For this, three main options are available (ignoring interpretation):

1. The compiler generates an object file containing machine code and meta-information, which is loaded into memory, relocated and linked with other modules by a module loader.
2. The compiler generates an object file containing intermediate code that is compiled into memory on-the-fly at load time [37].
3. The source code is syntax-checked and stored in tokenized form and then compiled on-the-fly at load time [63].

As each of the options has its own advantages and disadvantages, the runtime system should not exclude any of them in advance. Therefore, we separate the module loading mechanism into two parts: a basic part that handles the location and recursive loading of modules, and a plugin part that creates the runnable machine code in memory. The plugin could simply load a compiled object file, or it could perform on-the-fly compilation.

The plugin part of the module loader uses the basic part to obtain information about procedures, types and variables exported from modules, so that it can link the newly loaded module with the modules that it imports. In a similar way, the plugin links the module with any runtime facilities that it may require.

4.2.2 Object Model

Active Oberon is an object-oriented language and programs written in it are mainly concerned with creating and manipulating *objects*. The underlying object-oriented characteristics of Active Oberon, supported by the runtime system, are summarized here.

An *object* is an anonymous, dynamic instance of an *object type* (i.e., *class* in object-oriented terminology), referenced by at least one object reference variable, which is a typed pointer variable that stores a reference to an object compatible with its type.

An object contains *fields*, which are named variables nested in the scope of the object and store its state. An object also has *methods*, which are named procedures nested in the object scope that can manipulate the state of the object.

Object types can be *extended* (single inheritance) by defining a new object type as an extension of an existing one. The fields and methods of the existing object are *inherited* by the new object. A method of the new object type can *override* a method of the same name in the existing object type. In this case the method in the existing object is replaced by the new implementation in the new object, which must have the same signature. It is possible to perform a *supercall* from an overriding method to the overridden method. A *type test* can be used to test whether an object reference variable points to an object of a specific type.

An object is allocated explicitly (e.g., with the Active Oberon NEW procedure), but is deallocated automatically by the system when no references to it exist any more. In practice, the deallocation can be delayed until the system's garbage collector has completed its next full cycle.

As these characteristics are essentially the same as in Oberon-2 [71], this part of the runtime system is based on the ETH Oberon systems, specifically Native Oberon.

4.2.3 Process Model

The major addition in the Active Oberon object model is the object body, which specifies the inherent activity of an active object. As it does not make much sense to execute the activities of active objects sequentially, the runtime system must provide concurrent *processes* for this purpose.

To ensure that hundreds or even thousands of active objects can be created, the processes should be suitably lightweight. This can be ensured by only associating a small process descriptor and a stack for the local variables of the executing program with a process. Processes should not be used as domains or containers for resource allocation purposes and they all share the same global linear address space. When aiming for a minimal runtime system, every concept added that may increase the system complexity should be considered carefully.

Scheduling issues. In a multiprocessor system, different processes should obviously be assigned to different processors, but each processor can execute only one process at a time, and typically there may be many more runnable active objects than processors. In a generic population of possibly independent active objects, it is likely that the processes of some objects will perform computations for extended periods of time, without any natural reason to pause their execution. If more such processes exist than there are processors available, they will block the execution of others in the population. Unless further measures are taken, only some processes will make progress during these extended periods of time. Therefore, some form of *pseudo-parallelism*, in which a processor interleaves the execution of several processes, is required.

A possible solution is to require computationally intensive active objects to release the processor voluntarily by calling a runtime procedure from time to time. However, this solution is not satisfactory; experience with tasks in Oberon have shown that this is difficult to coordinate, especially when active objects make use of other objects to perform their work. It can also be inefficient, as releasing the processor too often causes unnecessary overhead, and releasing it too seldom restricts the progress of other active objects.

A better solution is to use *time-slicing*, i.e., let the system automatically preempt long-running processes by setting a timer to interrupt them. In this way the system is in control of the frequency of process switches and the active object programmer does not have to worry about releasing the processor.

To make the Aos runtime system viable for realtime environments and responsive in interactive use, a facility must be provided to allow more important processes to run before less important ones. Therefore the assignment of relative *priorities* to processes is foreseen. At any time, the processes executing will have a priority not lower than that of all other runnable processes.

There are basically two approaches to scheduling multiple processors: either each processor is responsible for its own scheduling, or one processor is responsible for scheduling on all the processors [112]. The first approach is preferred for its simplicity, symmetry and scalability.

Synchronization. Runtime support for active object process synchronization has two aspects: the exclusive regions of active objects have to be protected by a *locking mechanism*, and the await statement requires *condition management*, which is the periodic evaluation of synchronization conditions specified by the programmer (cf. 2.2).

The compiler and runtime system use *locks* (binary semaphores) to ensure that the exclusive regions of an object are entered by at most one process at a time. Every object (or module) instance is allocated one lock, which is acquired by a process entering the exclusive region, and released when the process reaches the end of the region. A process is not allowed to recursively acquire the same lock.

For well-behaved synchronization, the compiler and runtime system enforce the rule that an await statement must always be enclosed in an exclusive region. When await is executed, the current process must hold the lock of the enclosing object (or module), or a runtime error occurs. When the condition is found false and the current process has to be delayed, the lock is released first. Likewise, the object state modification that establishes the condition being waited on must be made in an exclusive region of the same object. This obligation is the responsibility of the programmer and is not automatically checked. It is also assumed that the conditions are free of unwanted side effects, but this is not enforced by the system.

The conditions that delayed processes are waiting on have to be evaluated periodically by the runtime system. Under the assumption that the relevant object state is only changed in an exclusive region, it suffices to evaluate the conditions related to a specific object instance every time a process exits the relevant exclusive region.

In the case where a process exits an exclusive region, and the system finds that the condition of a waiting process is now true, the lock is not released, but instead transferred atomically to the process that was waiting before. That process is then activated and continues to run as soon as it is scheduled on some processor. In this way, if one process establishes a condition B, the woken process can safely assume that B still holds when its await statement terminates (e.g., in figure 2.3, when Put establishes $n \neq 0$, Get can safely assume this still holds when its await statement terminates). In other words, the following proof rule holds for await, where $\mathcal{I}$ is the invariant of the associated exclusive region:

$$\{\mathcal{I}\}\ \mathsf{AWAIT}(B)\ \{\mathcal{I} \wedge B\}$$

It is of course possible for several processes to be awaiting conditions in the same object context. When a process exits the relevant exclusive region, the runtime system evaluates all the waiting conditions in sequence until the first true condition has been found, and the associated process is reactivated. It is not necessary to evaluate the rest of the conditions immediately, as they will be checked when the reactivated process exits the exclusive region.

Context switches. When implementing pseudo-parallelism, it has to be decided when a processor will perform a *context switch* from one process to the next. There are a few situations where this can be done naturally, because the running process has no useful work to do any more. These are called *synchronous context switches*, because they are synchronized with the actions of the running process. They occur when the running process:

1. attempts to enter an exclusive region held by another process,
2. executes an await statement with an unsatisfied condition, or
3. terminates.

A synchronous context switch is always the result of a call to the runtime system by the running process itself. This knowledge can be used to optimize these context switches by only saving the relevant subset of the processor state, similar to what is done with the coroutine library implementation in Modula-2 [131].

Time-slicing and process priorities provide two more reasons for a processor to be switched away from a running process, namely:

1. it has executed continuously for an extended period of time and has used up its time slice, or

2. a higher-priority process becomes ready to run.

These two conditions, together with the three listed earlier, represent all possible reasons for a context switch to occur.

When a timer interrupt expires the time slice of the running process, or a device interrupt enables a higher-priority process to run, the result is always an *asynchronous context switch*, because the running process is preempted at an arbitrary location during its execution. In this case the full state of the process has to be saved.

In contrast, a context switch due to a higher-priority process becoming available can be either synchronous or asynchronous. Such a switch occurs when a process P that has been waiting on a lock or condition is suddenly able to run again, because some other process Q has released the associated lock. There are two situations to consider in this case. First, if Q has a lower priority than P, the situation is handled like a synchronous context switch, because Q has performed a system call to release the lock. The process Q is preempted, and process P starts running on the same processor. Conversely, if Q does not have a lower priority than P, it must continue to run on the same processor, and P must be scheduled to run on another processor that is currently executing a lower-priority process. The second case is more complicated to handle than the first, because it involves finding another processor running a lower-priority process, and preempting the process running on that processor.

4.2.4 Runtime System Calls

The support for active objects can be implemented by a handful of procedures exported from the runtime system: CreateProcess, Lock, Unlock and Await (cf. 5.9.3, 5.10 and 5.11). The compiler inserts normal calls to these procedures in the object code, and the module loader links these calls with the runtime modules.

To instantiate an active object, the compiler generates a CreateProcess call that creates a process to execute the object body. In the case of Active Oberon, active objects are instantiated with the NEW standard procedure call. This is compiled into: a call to allocate the object's memory, a call to the object's initializer, if any, and a call to create the process that will execute the body. Due to type extension, an object may have several active bodies, so a process is created for each of these.

When compiling exclusive regions, the compiler generates a Lock call at the start of the region, and an Unlock call at every point control leaves the region. For both calls, the self reference of the relevant object is passed as the only parameter.

The await statement is supported by the Await system call, which has three parameters. The first is a reference to a function that evaluates the associated condition being waited on, the second specifies the stack context of the condition and the last is the self reference of the relevant object. The condition function is a separately-callable function procedure generated by the compiler for every await statement. This function has one parameter that specifies the context of the statement and returns the result of the condition evaluation as a boolean value. It can be called by the runtime system when the condition needs to be evaluated.

4.3 Native System Facilities

The active object runtime system realizes the language support described above natively on the hardware without any operating system underneath. For example, memory management is handled directly by the runtime system kernel.

In fact, facilities are provided for implementing operating system services on the runtime system. This allows a lean and efficient system to be configured that excludes some or all of these services. For example, it would be possible to use the kernel to build a lean active object-based control application that requires little or no operating system support like a file system, a viewer system or user interface. In this way, control applications can be written in a language with active objects and garbage collection and compiled to run almost directly on the hardware, with just the small active object runtime system underneath.

A native system must include its own operating system services and device drivers and the latter require an interface to the underlying hardware. The Active Oberon language provides some low-level features for this purpose in the SYSTEM module (cf. 2.2.4), but this must be complemented by runtime system support.

Section 4.3.2 describes the Aos device driver model, which treats device drivers as normal programs, and section 4.3.3 describes the interrupt handling model, which maps interrupts to Active Oberon language constructs like procedures and objects.

4.3.1 Memory Management

One of most basic tasks of an operating system is the management of memory. In Aos, this is the responsibility of the active object runtime system itself, as no underlying operating system is used. Put simply, the task is to implement the Active Oberon built-in procedure NEW for the allocation of heap objects (active objects, normal objects and dynamic arrays). This includes the allocation of stack space for the local variables of processes. No explicit deallocation procedure is foreseen; the system has to deallocate unused memory automatically, with some form of *garbage collection.*

The Native Oberon system serves as an example of how simple memory management can be made. Virtual addresses are mapped identically to physical addresses (except for one area that is left unmapped to trap NIL pointer references), and the resulting memory is divided into two areas, one for the system stack, and one for the heap. This model simplifies device driver programming, as drivers can use virtual addresses directly when performing direct memory access (DMA). As all allocated memory is physically contiguous, drivers can work directly on data shared with their client programs, without having to resort to copying or complicated page mapping.

The design of Aos aspires to this ideal, but there are complicating factors. Below, some assumptions are stated and justified, and the resulting issues discussed. An implicit assumption in an extensible system (cf. 3.1) is that a single global address space is used, allowing data to be shared directly between different parts of the system. This does not necessarily exclude the use of different protection domains [84].

No object copying. Since Aos is intended for inherently cooperative environments (cf. 3.1), it is assumed that the range checking and type safety provided by the language are sufficient tools for maintaining system integrity, and that separate address spaces are not required for this purpose.

The runtime system and system services sometimes need to bypass the type system and the associated range checks, to access memory directly. Judicious use of this technique in performance-critical parts of the system (e.g., device drivers and system libraries) can improve performance significantly, which can be justified, even given the additional risk of compromising system integrity and reliability. But, when the type system is bypassed, the objects being operated on must remain at the same virtual address during the whole operation.

A copying (or compacting) garbage collector needs to copy objects to different virtual addresses during their lifetime. As long as the object is accessed only in a type-safe way via explicitly declared pointer variables, the runtime system and compiler can cooperate to ensure access consistency.

Copying garbage collectors have three apparent advantages [55]. Firstly, allocations are cheap; secondly, fragmentation is avoided; and thirdly, the cache locality of data is possibly improved. However, copying data around all the time seems wasteful, and the advantages are not conclusive. Zorn found that a generational mark-and-sweep collector can be more effective than a copying collector [138], and Boehm et al. have developed a very effective conservative mark-and-sweep collector, which reduces fragmentation with a two-level allocation scheme [13, 14].

Under these considerations, the use of a copying garbage collector is excluded to simplify the design.

No demand-paged virtual memory. A *virtual memory* system uses demand paging to back up parts of the *virtual address space* to secondary storage. This means that virtual pages are sometimes not physically present, and do not always map to the same physical pages. The NEW procedure initializes a pointer variable with the virtual address of the memory allocated. Normally, the actual physical location of the memory is not relevant to a program; it could be allocated on non-contiguous physical pages, or paged out to secondary storage.

As an exception to this, device drivers performing DMA have special memory allocation requirements. The memory used has to be physically present at the same location during the whole transfer and, in most cases, there are also special alignment and contiguity restrictions. In spite of this, device drivers in Aos should ideally be completely normal programs, and should be able to act directly on the data shared with their clients without special allocation procedures or additional copying.

For this reason, and based on the experience with Native Oberon, it was decided not to consider virtual memory in the system design. Although virtual memory allows programs to handle more data directly, the sometimes severe and unpredictable performance degradation is unsatisfactory. Virtual memory presents a false abstraction, because of the extreme latency of accessing paged-out memory. It is better to handle data on secondary storage explicitly, by using files. As an aside, the availability of apparently unlimited virtual memory on some operating systems seems to encourage wasteful design.

The assumptions above allow the arrangement of the heap as a single shared area in the virtual address space, mapped directly to physical memory as in Native Oberon. Were it not for the requirements of process stack management (and to a lesser extent, NIL reference checking), it could even be worth considering the elimination of the virtual-to-physical mapping completely.

Process stacks. The Native Oberon stack model can unfortunately not be used for Aos, as it must be able to manage the process stacks of hundreds of concurrently running active objects, not just one. The stacks must also be able to grow dynamically, since for general programming tasks it is not possible to predict in advance how large they should be.

Conceivably, a process stack could be allocated as a contiguous block in the normal heap, with compiler-generated stack overflow checks (in the absence of a hardware stack limit register). When a stack overflows it could be reallocated by allocating a larger area, copying the existing stack, and modifying all references to the stack, including the dynamic and static link chains. This would require detailed meta-information describing the location of reference parameters, as local variables can also be referenced indirectly. It would also require integration with the garbage collector, which has to scan the stacks for roots. Such an ambitious scheme requiring close cooperation between the compiler and runtime system would probably be fraught with unforeseen complications.

Brinch Hansen described an interesting stack allocation scheme for parallel programs, which interleaves fixed-sized stack segments from different processes in the heap. Unfortunately the solution is “not realistic for an operating system, which allocates an unbounded number of segments, most of which are unique to particular user jobs" [21]. On a multiprocessor, the scheme also requires two lock and unlock operations per block activation, which would be an intolerable overhead.

Another option for managing process stacks is to allocate a fixed area of the virtual address space for each stack. Initially, only a small part of the area, typically one page, is allocated physically. When a process uses more than the allocated space, a page fault occurs and the system allocates more physical pages to the stack. This solution is easy to integrate with the garbage collector, which has to scan the stack areas (possibly conservatively) for root pointers.

Direct memory access in device drivers again presents a problem. Since the physical pages of a virtual stack are not necessarily contiguous, a driver can not always perform direct memory access on stack pages. However, this problem can be avoided by allocating buffers in the heap instead. This solution is not entirely satisfactory, but workable, because DMA buffers tend to be large, and it is therefore better not to allocate them on a stack with limited size.

The virtual stacks organization has the disadvantage that the maximum size of a stack is limited a priori. Depending on the size of the address space, it might also place an unsatisfactory limit on the total number of process stacks. Nonetheless, this organization was chosen for Aos, because of its straightforward implementation and compiler independence.

4.3.2 Device Driver Model

Unlike closed systems (cf. 3.1), where device drivers normally reside in the kernel and have special restrictions, the aim in Aos is to make device drivers as similar to normal programs as possible. A device driver is just a normal module that exports some service functionality abstracting a device and uses low-level facilities of the language and kernel to communicate with the hardware. Specifically, it shares the same address space with its client modules and can therefore work directly on client parameters without any additional data copying.

The fact that device drivers share the same address space with user programs does not have any intrinsic effect on the stability of the system. Even on systems where device drivers reside in a separate address space, an error in a device driver can cause a system crash, as they use low-level facilities like DMA that bypass the normal processor checks. What is more important is that device drivers have a simple and unexceptional programming model, so that errors are less likely.

The Oberon system uses a *polling I/O model*. A device driver for an input device provides a polling procedure and a read procedure. The Oberon main loop or a user program polls to determine if input is available, and, if so, it calls the read procedure, which will return immediately. For output, a write procedure is provided that sends the given data and returns when done. This model is simple and efficient in a single-process system, but inappropriate for a multiprocessing system. If a process is polling a device continually, it is wasting processing time that could be used by other processes to do useful work.

Instead of polling, Aos device drivers should provide blocking interfaces, where an I/O request to a driver suspends the client process until it can be satisfied. While the requesting process is suspended, other processes can perform useful work. This kind of interface can be realized by encapsulating the driver in an object, and using the await statement to block the client process in the object until it can continue. This usually requires synchronization with an interrupt from the device.

4.3.3 Interrupt Handling Model

An interrupt is usually used by a device to signal the completion of some I/O operation, and must therefore be routed to the relevant device driver.

Low-level interrupt handling. The Native Oberon system uses a *procedure call model* for handling interrupts. A device driver can register an Oberon procedure as an interrupt handler for a specific interrupt. When the interrupt occurs, the kernel saves the processor state and calls the registered procedure. All interrupts are disabled for the duration of the procedure, and when it returns, the processor state is restored so that execution continues from the interrupted location. Interrupt handlers may re-enable interrupts, and can then be interrupted by higher-priority interrupts, according to the priority model defined by the interrupt controller.

The polling and read procedures provided by a device driver communicate with the interrupt handler via shared global variables. These procedures typically disable interrupts for a short time while accessing the shared variables, to ensure that no conflicting concurrent accesses are made by the interrupt handler.

The procedure call model has very little overhead. Some minimal 'glue' code is needed to save the processor state and call the registered procedure, and afterwards acknowledge the interrupt and restore the processor state. On Ceres Oberon this code is generated inline by the compiler for even less overhead (no kernel involvement during the interrupt). This is not done here, in order to decouple the compiler and runtime system more, thereby simplifying the support for other languages than Active Oberon.

At a low level, Aos too uses this simple procedure call model for handling interrupts. However, interrupt procedures are restricted in what they can do. For example, they are not allowed to enter exclusive regions or use await statements, as they are not independent processes that can be suspended. To treat device drivers as first-class objects, a higher-level interrupt handling concept is required.

Exception handling. At the processor level, *exceptions* (i.e., traps) are similar to interrupts, but are caused by errors during the execution of a processor instruction (e.g., a division by zero), or by special exception-generating instructions (e.g., software interrupt instructions). An exception usually signals a failure in the currently running program, therefore a typical response is to terminate the current process.

In Native Oberon, all exceptions are handled by the system, not by user programs. The default action is to display a *trap text* with a symbolic stack traceback showing the context of the exception. Then the stack is reset and the Oberon main loop is restarted, thereby aborting the current command or task. Although it is possible for a user program to install an alternative system-wide exception handler, the default handler is used in most cases.

Since the Active Oberon language has no general exception handling facilities, e.g., an equivalent to the try-catch construct of C++, the runtime system does not have to provide complicated exception handling support. When an exception occurs, the kernel simply terminates the current process, or restarts it if the SAFE modifier was placed on the active object body that started the process (cf. 2.2.4).

To allow other language environments running on the kernel, e.g., Java, to handle their own exceptions, it suffices to allow each process to install an exception handling procedure, similar to an interrupt handler procedure. A plugin module that provides support for such a language can install a handler that gets control when an exception happens and can unwind the stack or do whatever is necessary to handle it.

Active object interrupt handling. The higher-level interrupt handling mechanism referred to above is described here. It allows device driver objects to directly handle interrupts and synchronize with their clients using exclusive regions and await statements.

Figure 4.2 shows the outline of a device driver object using this mechanism. The example driver is for a device that periodically produces data and causes an interrupt to signal its availability. The interrupt handler reads the data from the device and stores it the object's fields, where a client can fetch it using the Read method. This method uses an await statement to wait until data is available.

The example object is an extension of a system-supplied abstract base object InterruptHandler, which has a HandleInterrupt method overridden and implemented by the driver. The driver object registers to receive interrupts by making a special system call in its initializer.

The implementation of interrupt handling is complicated by the fact that the same interrupt number can be shared by different devices. The PCI bus standard [82], used in many modern system architectures, allows independent devices to share level-triggered interrupt lines. The kernel must allow multiple processes to register for the same interrupt. When the interrupt occurs, all the registered handlers must be activated, and only when the interrupt has been processed by all, can it be unmasked again. The device drivers must handle the resulting spurious interrupts gracefully.

To implement the high-level interrupt handling mechanism, the system provides an active object behind-the-scenes for every interrupt number. When an interrupt occurs, the process of this object is scheduled to call the registered interrupt handler methods. It sequentially calls all the methods registered on the specific interrupt. The interrupt is masked while these calls are being performed and after they have all returned, the interrupt is unmasked and the system process is suspended until it is signalled again.

The high-level interrupt handling model described here was imple-

```
TYPE
  Driver = OBJECT (InterruptHandler) (* extends system object *)
    VAR "driver state"(* shared by client and interrupt handler process *)

    PROCEDURE Read(...); (* client process calls this to read data *)
    BEGIN {EXCLUSIVE} (* mutually exclusive with interrupt handler *)
      AWAIT("driver state indicates data available");
      "return state information"
    END Read;

    PROCEDURE HandleInterrupt; (* system calls this on interrupt *)
    BEGIN {EXCLUSIVE} (* mutually exclusive with client *)
      "get data from device and possibly modify driver state"
    END HandleInterrupt;

    PROCEDURE &Init; (* initializer *)
    BEGIN
      "register SELF as interrupt handler in system"
      "initialize device"
    END Init;
  END Driver;
```

Figure 4.2: Active Oberon pseudo-code of a device driver.

mented in the Aos kernel. The outline of an alternative model that was not implemented, is presented in appendix C.

Chapter 5

Multiprocessor Runtime Implementation

This chapter describes the native implementation of the active object runtime system (the bottom layer in figure 3.1) on the Intel IA-32 symmetric multiprocessor (SMP) architecture.

5.1 Multiprocessor Implementation Issues

Target architecture. The active object runtime system was first implemented on the Intel IA-32 SMP architecture realized in the Intel P6-family of processors — the Pentium Pro, Pentium II and Pentium III [51]. This architecture was selected as we aimed to show how a lean but powerful multiprocessor system can be developed for relatively low-cost multiprocessor machines available today. The implementation also works with other IA-32 processors like the Pentium and the Pentium 4, but does not use the new multiprocessor features introduced on the latter. It would be relatively easy to adapt it for i386 and i486 embedded processors.

The Intel architecture (see figure 5.1) is typical for multiprocessors with up to eight processors. Each processor is tightly coupled with its primary and secondary cache, which connects it to the shared memory subsystem via the system bus. Each processor has an advanced programmable interrupt controller (APIC), which connects it to a special bus for interrupt delivery and interprocessor communication. This bus

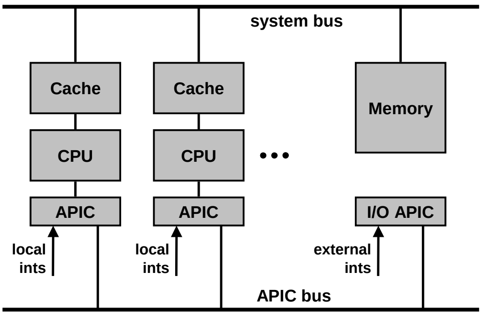


Figure 5.1: Intel SMP multiprocessor architecture (simplified).

is connected to an I/O interrupt controller (I/O APIC), which handles interrupt communication with external devices.

The processor provides three mechanisms for atomically operating on the shared memory: primitive atomic operations, bus locking and a cache coherency protocol. Some primitive memory operations (reads and writes) are guaranteed to be atomic: 8-bit memory accesses, aligned 16-bit memory accesses and aligned 32-bit memory accesses. The processor automatically asserts a special bus lock signal during some critical operations, e.g., when executing an exchange (XCHG) instruction or when updating page table entries. Some instructions may be prefixed with a lock prefix to assert the same signal under software control. In this case, if the relevant data is cached, the bus signal is generally not asserted, but the locking is applied on the cache instead, and the cache coherency protocol is relied on for atomic access. This protocol prevents different processors from simultaneously modifying the same cached memory area.

Memory ordering model. The local processor caches significantly improve program performance, but also complicate programming due to their imposition of a possibly non-intuitive, relaxed memory ordering. The memory ordering model is an artefact of the buffering and caching implementation of the processor's memory interface [33, 69]. Put simply, the interaction of shared memory reads and writes from the same or different processors may lead to unexpected results.

It should be noted that the memory ordering model is only relevant to user programs when processes interact without acquiring critical section locks, e.g., when a shared variable is accessed without protecting it with exclusive regions. This kind of access often occurs in performance-critical software, where avoiding a locking operation in the common case can provide great benefit.

Our approach to this problem is to simply expose the memory ordering model of the P6-family processors, since this model is still relatively intuitive. Our compilers align 8-bit, 16-bit and 32-bit variables so that they can be accessed atomically. If Aos is ported to other multiprocessor architectures with a more relaxed memory ordering, e.g., the IA-64 architecture, an approach where the compiler and runtime system cooperate to present a stronger model may be more appropriate [126].

Interrupt controllers. The local APIC built into every processor (see figure 5.1) is responsible for routing interrupts to it. These controllers are connected to a shared I/O APIC that routes interrupts from I/O devices to the processors. The routing can be static so that an interrupt is always routed to a specific processor, or it can be dynamic, so that interrupts are routed to the lowest-priority processor. For this case, every local APIC has a processor priority register that can be updated by software.

The APIC can be used by any processor to initiate an interrupt on any subset of the processors. This interprocessor interrupt facility is used by the runtime system to synchronize different processors, e.g., during garbage collection.

Another case where interprocessor interrupts are used is in the synchronization of the paging structures shared by all processors in Aos. When a page is unmapped from the virtual address space, (e.g., when a process stack is deallocated), an interrupt is sent to every processor requesting it to clear its translation lookaside buffer (TLB).

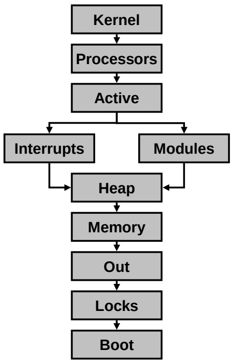


Figure 5.2: Aos runtime system module structure.

The APIC can also be programmed to generate periodic interrupts to the local processor. This facility is used by the runtime system to implement time-slicing and statistical profiling.

5.2 Runtime Module Decomposition

Figure 5.2 shows the module structure of the runtime system and table 5.1 shows the responsibilities of the various modules. The implementation of the kernel was split into many small modules to make it more understandable, even though it could probably have been made smaller by combining all kernel functionality in one module as in the Oberon system. This also has the consequence that some kernel modules (e.g., Heap) export implementation details. Where possible, modules above kernel level should only import the top-most kernel module.

The bottom two modules (Boot and Locks) can be seen as an extension of the processor's instruction set with facilities needed in the

Module	Responsibility
Kernel	Portable kernel interface
Processors	Multiple processors
Active	Active objects
Modules	Modules and types
Interrupts	Low-level interrupts
Heap	Heap and garbage collector
Memory	Virtual address space
Out	Serial console output (debugging)
Locks	Fine-grained locks
Boot	Boot loader and environment interface

Table 5.1: Aos runtime system module responsibilities.

implementation of the runtime system.

The Boot module acts as interface between the runtime system, the boot loader and the firmware, providing access to the machine's hardware configuration information. This is required since the hardware environment can be quite diverse and complex. It also assigns every processor in a multiprocessor system a unique sequence number, which can be used to index per-processor data structures, and provides some primitive atomic operations.

The Locks module provides locks for the protection of runtime data structures. As the critical sections in the runtime system are relatively short and do not contain await operations, the locks are implemented with busy-waiting.

The Out module provides console output and is used mostly for debugging purposes. During system startup some progress information is written to the console. This is invaluable to find startup problems.

The virtual address space and object storage management aspects of memory management are implemented in the Memory and Heap modules, respectively. The Memory module initializes the virtual address space using the paging facilities of the processor and manages page-level allocation and deallocation. It provides procedures for accessing memory-mapped devices and for allocating and deallocating process stacks and heap memory. The Heap module manages the heap, which is used to allocate objects, dynamic arrays and untyped blocks of memory using Active Oberon's NEW standard procedure call. Deallocation of unused objects is performed automatically by a mark-and-sweep garbage collector.

The Modules module is responsible for modules, types and commands. It implements the dynamic loader algorithm, while the actual loading of an object file is handled by a plugin.

Three modules are responsible for implementing active objects and interrupt handling: Processors, Active and Interrupts. The Interrupts module manages the interrupt controller and maps interrupts into Oberon procedure calls. The Active module is the heart of the runtime system. It implements lightweight processes for active objects, schedules the processes, manages the object locks and synchronization conditions and implements active object-based interrupt handling and timing. The Processors module implements multiprocessor support. It is responsible for interprocessor communication and booting additional processors, when present. It also provides a periodic timer facility used for accurate timing in device drivers and for time-slicing, which is essential on singleprocessor machines.

The Kernel module exports primitive atomic operations, fine-grained timing and finalization operations (cf. 5.6). In some cases it also re-exports some functionality from the rest of the runtime system, where this can be done portably and with negligible performance loss. When higher-level modules need kernel services, they should import only this module, as far as possible, to be less dependent on a specific runtime implementation. It is intended as the only module in the runtime system with a portable interface that can be considered stable, even when the implementation of the runtime system changes. This gives a system implementor the freedom to change the interfaces of any other runtime modules, without adversely affecting many higher-level modules.

The following sections of this chapter describe the modules of the kernel in a bottom-up fashion.

5.3 Interfacing with the Environment

The first piece of software to get control when an Intel IA-32 machine boots is the firmware (i.e., BIOS) provided by the manufacturer. This program performs some initial testing and initialization of the processors, memory and boot devices, and proceeds to boot the installed operating system kernel. It provides a level of manufacturer-independence, allowing the kernel to boot on machines with different motherboards, memory configurations and boot devices.

The modules of the Aos kernel are statically linked into a *boot file*, which is written to a hard disk or other boot device, together with a boot loader. After initialization, the firmware loads the boot loader into memory and starts executing it. The Native Oberon boot loader is used, which performs the following tasks:

1. Set the segment registers and stack for a simple 16-bit execution environment which allows us to use firmware services like the boot device driver.

2. Read the boot file into memory and check its consistency.

3. Query the firmware for some hardware information and store this in memory for later use.

4. Set up a simple 32-bit execution environment, move the boot file to its final location at address 1M and start executing it.

Boot module. The first module in the Aos kernel that gets control is the bottom-most module, called Boot. It is responsible for low-level initialization and interfacing with the firmware, which makes available some information about the hardware environment, e.g., the number of processors available. Information which is not reliably provided by the firmware, e.g., the total memory size, is computed here.

The module exports some functions for use in the rest of the kernel, the most important of which is the processor ID function, which returns a unique sequence number for each processor it executes on. This number is used to index arrays containing per-processor data. Also exported are an atomic test-and-set operation and atomic versions of the INC, INCL and EXCL standard procedures of Oberon, used when operating on shared data.

When this module gets control, the processor that has been elected as boot processor is running, and the other processors are disabled. After this module has initialized, the boot processor continues executing the body of the next-higher kernel module, and so on until the Processors module which finally boots the other processors (cf. 5.12).

5.4 Protecting Runtime Data Structures

The data structures of the runtime system have to be protected from concurrent access by different processes. On a multiprocessor system, processes running on different processors can make parallel calls to the runtime system. Additionally, if processes are allowed to be preempted while they are executing in the runtime system, the data structures have to be protected against pseudo-concurrent accesses, which happen when a process switch occurs in a critical section.

Examples of such data structures are the queues and arrays mentioned in section 5.9.2, but also the structures of the memory manager, interrupt handling, module management, and other parts of the runtime system.

On a singleprocessor system, the critical sections can be protected by simply switching off interrupts during their execution. This disables process switches, ensuring that the current process has exclusive access to the data during the critical section. The Oberon system uses this solution to protect data structures shared between the Oberon process and interrupt handlers in device drivers. This is not sufficient in a multiprocessor environment, because different processors can access shared data even when they are running with interrupts disabled.

Spin locks. Critical sections in a multiprocessor environment can be protected by binary semaphores implemented with busy-waiting — also known as *spin locks* [7]. Most modern processors have primitive atomic instructions such as 'test-and-set' or 'exchange' which can be used to implement spin locks. The lock is implemented as a boolean variable. The *acquire* operation sets the lock variable to true, and at the same time tests its existing value. If the value was true, the operation is retried (and the value remains true). Otherwise, the operation finishes with the value set to true, indicating that the lock is held, and that the critical section may be entered. The *release* operation at the end of the critical section simply sets the lock value to false, allowing other acquire operations to succeed.

Spin locks as described above can interact with interrupts to produce deadlocks. If a critical section is interrupted, and the interrupt handler attempts to acquire the lock that protects this critical section, it will wait forever. This is avoided by disabling interrupts during critical sections.

Another reason for disabling interrupts during critical sections is to avoid interrupts that could cause the process that is currently executing in the kernel to be preempted and rescheduled on another processor. This is important as kernel critical sections often manipulate per-processor data structures and therefore assume that the same processor executes the whole critical section.

When protecting critical data structures, there is a trade-off in deciding how large a critical section to protect. Using few locks reduces the complexity and makes it easier to avoid deadlock, but also reduces the potential parallelism. Using many locks improves the potential parallelism, but makes it more difficult to avoid deadlocks. The latter solution may also have a higher overhead due to processes having to acquire and release more locks.

As an example, the runtime system could use one single lock to protect all its data structures. This implies that only one process can execute in it at any time, which could severely restrict parallelism. A better option is to have a lock for every data structure, but partitioning the critical sections is not straightforward when the data structures — or the interactions between them — are non-trivial.

A middle road taken in Aos is to structure the runtime system as several modules, each of which manages a few related data structures, and to use a single lock for each module. This makes it relatively straightforward to show that an implementation is deadlock-free.

Locks module. The handling of spin locks is concentrated in the Aos kernel module called Locks, which provides two kinds of spin lock. The first kind is used to protect global kernel data structures and the second kind is used to protect the header fields of an active object (cf. 5.10.1). The difference between the two kinds is that the first also disables interrupts during the critical section and therefore needs to keep track of the nesting level of locks. The second kind of lock is a straightforward lightweight spin lock which can not be nested.

The following strategy is used to avoid deadlock in the kernel: As far as possible, only a single lock is held at a time. For the cases where more than one lock is required simultaneously, the locks are acquired in a fixed order, so that no hold-and-wait cycles can occur. To simplify the issue of lock ordering, only one lock is used per module, to protect all the shared global variables of the module at once. This lock is acquired at critical entry points to the module, and released at exit points. There are three kinds of entry points: exported procedures, local procedures that are registered as upcalls in other modules (e.g., interrupt handler procedures) and overridden methods of objects from other modules (rare in the kernel). Procedures that are not module entry points do not need to acquire the lock, as they are always called from other procedures that are entry points. If we further ensure that no locks are held when an upcall is performed, it suffices to order the locks according to their level in the module hierarchy. Thus the acyclic module import structure is used to order the locks and avoid deadlock, as in the THE system [29].

To evaluate the cost of locking, especially the decision to use only one lock per module, the locking module optionally accumulates the following statistics on the use of the individual locks: the number of times a specific lock was acquired, the mean and maximum time the lock was waited for and held, and the variance of these times.

5.5 Memory Management

Section 4.3.1 presented the design decisions behind the Aos virtual address space organization and concluded that a single shared virtual address space containing the heap and process stacks should be used, with the heap area mapped identically to the physical address space. In this section we first summarize the memory management options of the IA-32 architecture and then show how the virtual address space is mapped to the physical address space.

IA-32 memory management options. The IA-32 architecture [53] supports segmentation and paging. Paging can be switched off, but segmentation is always enabled. The processor translates 48-bit *logical addresses* via segmentation to 32-bit *linear addresses*, which are then optionally translated via paging to 32-bit or 36-bit *physical addresses.* Pages can be 4KB, 2MB or 4MB big, and can be mapped to physical memory or marked as not-present. To speed up address translation, page-table entries are cached in code and data *translation lookaside buffers* (TLB caches).

A logical address consists of a 16-bit segment selector, specifying some segment, and a 32-bit offset into the segment. The segment selec-

Feature	Introduced	Page sizes	Levels	Address size
Paging	Intel386	4KB	2	32-bit (4GB)
PSE	Pentium	4KB/4MB	2/1	32-bit (4GB)
PAE	Pentium Pro	4KB/2MB	3/2	36-bit (64GB)
PSE-36	Pentium III	4MB	1	36-bit (64GB)

Table 5.2: IA-32 architecture paging options.

tor is usually specified implicitly via one of the 6 segment registers, and is an index into the *global descriptor table* (GDT) or a *local descriptor table* (LDT). These tables specify the 32-bit linear base addresses, sizes and other attributes of the segments in the system. A segment can have any byte-granular size up to 1MB, or any 4KB-granular size from 4KB to 4GB. Segments are typically used for general-purpose data, but some special segments describe system data structures (local descriptor tables, task-state segments, call-gates, interrupt-gates, trap-gates and task-gates).

Table 5.2 summarizes the paging options present on IA-32 processors. All the listed processors are backward compatible with earlier processors in the table. On the Pentium, 4KB pages can be mixed with 4MB pages. The advantage of using large pages is that single-level page translation is performed and TLB utilization is improved.

Linear addresses are limited to 32 bits, therefore more than 4GB of physical memory can only be accessed using paging. The PSE-36 feature of the Pentium III provides a direct way to do this, by allowing 4MB pages to be located on any 4MB boundary in the 64GB physical address space. Unfortunately, it does not allow mixing of 4KB and 4MB pages. The PAE feature of the Pentium Pro also enables a large physical memory, using a triple-level page translation structure with mixed 4KB and 2MB pages.

IA-32 physical address space. Figure 5.3 shows the physical memory layout of an IA-32 machine. It can be seen that the RAM is not physically contiguous, as there are fixed areas reserved for I/O devices (between 640KB and 1M, and above address X) and firmware ROM.

Due to backward compatibility issues with old IBM PC/AT-class machines, the RAM at 512K and 15M is optional and can be masked out by the system to allow older I/O devices to occupy these areas. The

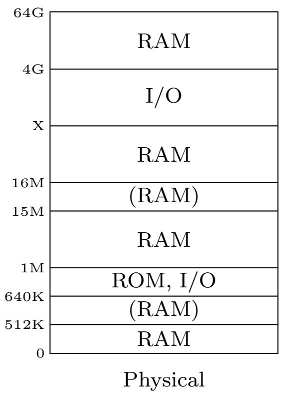


Figure 5.3: Typical IA-32 system physical memory layout.

Aos implementation assumes that these addresses contain RAM, which is true on most modern machines.

The address X designates the highest RAM address below 4G, and depends on the amount of installed RAM. Its largest possible value is 4076M, since at least 20MB is always reserved for I/O devices at the top of the 32-bit address space. The RAM above 4G is only present on very large systems that have at least 4GB RAM.

Each processor starts up in 16-bit x86 mode with paging disabled, and can therefore only access the low 1MB of physical memory. After switching to 32-bit mode in the boot loader, the whole 4GB address space becomes accessible. The memory above 4G is never directly addressable, and can only be accessed via paging.

Memory module. The Memory module initializes the virtual address space and provides procedures for managing process stacks, allocating heap memory and making memory-mapped devices accessible.

When the module gets control the boot processor is initialized to run in 32-bit flat segmentation mode with paging disabled. Flat mode uses two segments in the GDT, one for code and one for data, that are

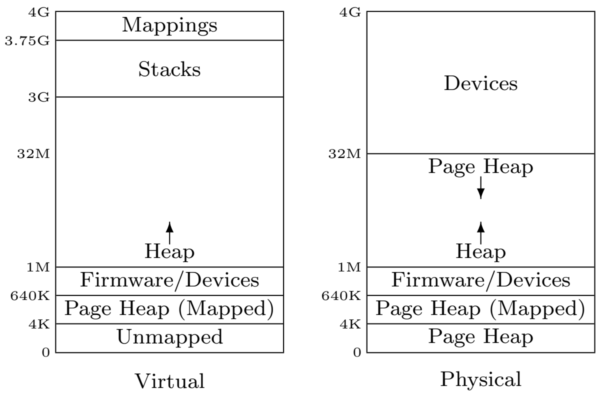


Figure 5.4: Address space organization (32MB RAM shown).

both based at address 0 and encompass the whole 4GB linear address space. All the segment selector registers are set to these segments, and in effect segmentation can now be ignored.

Figure 5.4 shows the main areas of the virtual address space, which are mapped to the physical address space using paging.

The *heap* starts at 1M and grows upwards. Virtual pages are mapped directly to physical pages, which simplifies device drivers that perform direct memory access (cf. 4.3.1).

The *stack area* contains all process stacks and has a fixed size and location close to the top of the address space. When a process stack is allocated, a fixed part of the stack area is allocated to it. Initially, only one page is physically allocated, but when page faults occur in this area, additional pages are allocated. Arbitrary physical pages from the page heap described below can be mapped to the stacks, and the pages are not necessarily physically contiguous. The virtual address ranges for the stacks are managed using a fixed-sized block allocator implemented with a bitmap.

The *mapping area* maps in parts of the physical address space containing memory-mapped devices and is at the top of the address space.

The *low area* below 1M is a mixed bag. It is mapped directly to physical memory, except for the very first page, which is left unmapped to trap NIL pointer references. The RAM below 640K is part of the page heap and the part between 640K and 1M contains firmware and memory-mapped devices. As the processors boot in 16-bit x86 mode, some low pages are reserved during booting.

For the page mapping, 4KB pages are used. Since the page size and other implementation-specific information is not exported from the module, it would be relatively simple to use the Pentium PSE extensions to allocate 4MB pages to parts of the heap, thereby reducing the load on the TLB caches and improving performance.

Physical memory is dynamically allocated to the heap and the process stacks. The physical pages for the heap come from the area above 1M in the physical address space, as they are identically mapped. These are managed as a simple allocate-only stack which satisfies the need for contiguous physical memory.

The physical pages for the process stacks come from the *page heap*, which initially consists only of the RAM pages below 1M. Once these pages are used up, physical pages are taken from the top of RAM downwards towards the end of the heap. When a process stack is deallocated, its physical pages are returned to the page heap. The page heap is also used when allocating other physical pages, e.g., for page tables.

The single shared address space design is efficient and simplifies memory management, but it also constrains the RAM directly usable for program data to about 3GB excluding stack. Figure 5.5 shows how the organization would look on a system with 4GB RAM. The RAM above 4G is inaccessible, but could be made accessible through a window in the mapping area by using the PAE paging option. A good way to make this memory available to programs would be as a RAM file system or disk cache.

After the Memory module has initialized, the paging structures have been set up and the boot processor has switched from the boot loader stack and is running with paging enabled.

Memory module interface. The Memory module interface exports: heap size management for the object storage module (cf. 5.6), a stack data type for the processes module (cf. 5.9) and physical memory access procedures for device drivers.

The boot file which is loaded at 1M contains a static image of the

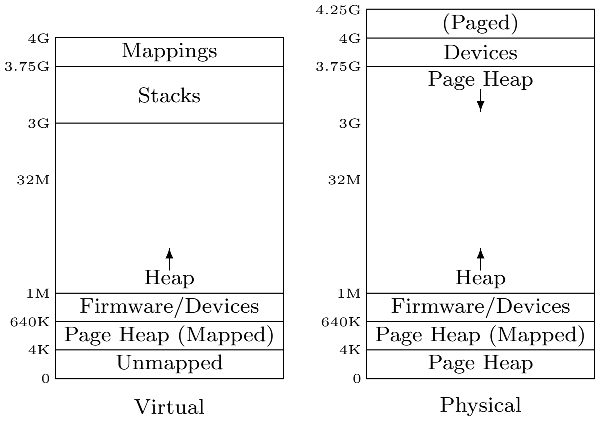


Figure 5.5: Address space organization (4GB RAM shown).

kernel modules in the same format as the heap managed by the heap module. This image forms the initial heap and the first procedure below returns its boundaries for the heap module. The second procedure allows the heap module to control the dynamic heap size. As input it accepts the requested heap end address, and as output it returns the new heap end address that was actually set.

```
PROCEDURE GetHeapAdr(VAR begin, end, first, free: LONGINT);
PROCEDURE SetHeapEndAdr(VAR end: LONGINT);
```

For stack management, the following type and operations are exported. A stack can be allocated or deallocated, and the virtual memory allocated to a stack can be extended.

```
TYPE
  Stack = RECORD
    low, high: LONGINT (* stack boundaries *)
  END;
PROCEDURE NewStack(VAR s: Stack; VAR sp: LONGINT);
PROCEDURE DisposeStack(VAR s: Stack);
PROCEDURE ExtendStack(VAR s: Stack; adr: LONGINT): BOOLEAN;
```

The first two procedures below allow a physical address range to be mapped into the virtual address space, or unmapped, respectively. The last function returns the physical address corresponding to a virtual address range, for direct memory access operations. If the specified address range is not physically contiguous (e.g., in the stack area), it returns −1.

```
PROCEDURE MapPhysical(phys, size: LONGINT; VAR virt: LONGINT);
PROCEDURE UnmapPhysical(virt, size: LONGINT);
PROCEDURE PhysicalAdr(virt, size: LONGINT): LONGINT;
```

The module also exports some initialization procedures used when booting other processors. The first procedure returns a low page containing a bootstrap loader for subsequent processors, and the second procedure is called by each processor to initialize its paging unit.

```
PROCEDURE InitBootPage(start: PROCEDURE; VAR phys: LONGINT);
PROCEDURE InitMemory; (* initialize paging *)
```

5.6 Object Storage Management

Heap module. The Heap module is responsible for managing object allocation and garbage collection in a contiguous growable block of memory obtained from the Memory module. Rather than implement a completely new allocation and garbage collection algorithm, it was decided to adapt that of Native Oberon, which is in turn based on earlier ETH Oberon versions. For Aos, the algorithm was extended to handle active objects and abstract root objects. The unsafe finalization mechanism was replaced by a safer one (p. 67). The allocator was adapted to share memory with the stack allocator. The heap management is implemented in a separate module to make explicit the interface between it and the rest of the runtime system and thereby simplify later replacement by another algorithm.

Oberon garbage collection overview. Ceres Oberon [134] uses a mark-sweep garbage collector [55]. The mark phase of the garbage collector is implemented without a recursion stack by using a *pointer reversal* algorithm [107]. The basic algorithm is essentially the same in most ETH Oberon versions, but later implementations lifted some restrictions and added support for object-orientation.

Ceres Oberon has two kinds of heap blocks: one for records allocated with NEW and one for untyped memory allocated with SYSTEM.NEW. Both kinds of block have a one-word header, which contains a mark byte for the garbage collector, and either the size of the block or a pointer to the type descriptor for a record. Such a pointer is also called a *type tag*, and is used to generate efficient *type tests*, and to find type information during garbage collection. The type descriptor contains the size of the record, the tags of its base types and the offsets of pointer fields in the record. It is allocated as an untyped heap block, therefore it does not need a meta-type descriptor. The fields of the type descriptor are used in type tests and in the garbage collector. As an implementation restriction, dynamic arrays are allocated as untyped blocks and are not allowed to contain pointers.

Heeb and Pfister [85] describe an improved algorithm that uses the type tag as an implicit mark byte. Templ [85] describes an extended algorithm that lifts the implementation restriction and allows dynamic arrays to contain pointers. Templ also developed a simple finalization mechanism [120]. For Object Oberon [70] and Oberon-2 [71] the type descriptor was extended to support *type-bound procedures* (i.e., virtual methods). The resulting heap block layout of later Oberon implementations is documented in [27, 117]. This was used with minor changes in Native Oberon.

As is mostly the case, the added functionality has a cost in terms of added complexity. In comparison with Ceres Oberon's two kinds of heap blocks, Native Oberon has four [25], and their structure is somewhat more involved:

1. Simple record block for dynamic records and objects.
2. Type descriptor block with meta-type descriptor for itself.
3. Array block for dynamic arrays containing pointers.
4. System block for dynamic arrays without pointers, module descriptors and untyped memory.

In Ceres Oberon, the garbage collector is invoked only from the main loop, when it is known that there are no heap blocks that are reachable exclusively from local pointer variables. Thus, local pointer variables on the stack can be ignored during the mark phase without risking dangling pointers, thereby simplifying the garbage collector. Unfortunately, this can lead to program failures due to an apparent lack of memory, when a command allocates a lot of temporary memory.

To solve this problem in a general way, later Oberon garbage collector implementations also consider local pointer variables on the stack as roots. To find such variables without additional stack layout descriptors, they scan the stack (and the processor registers) conservatively to find pointer candidates, and perform an additional sweep operation to confirm which candidates point to actual heap blocks. In this way the garbage collector can be invoked at any time — also from the allocation procedure when the free lists are empty. For Aos, this ability to invoke the garbage collector at any time is essential, as there is no steady state in which all process stacks are known not to contain pointer variables exclusively anchoring heap blocks.

Adding active objects. From the allocation perspective, an active object is similar to an Oberon-2 dynamically allocated record with type-bound procedures, except that it requires additional system data in every instance to manage locking and conditions. Therefore it is feasible to take the Oberon garbage collector and adapt it to support active objects. The main problem is how to extend the heap blocks of records to include the additional data. In Eamon [30], this is done by having the programmer declare every active object as an extension of a system-supplied base object that contains the necessary fields. For Aos this is considered too inflexible, since it would not be possible to extend an existing non-active object to make it active. In an object framework that supports active objects, every base object would have to be declared active, incurring system overhead on every instance.

The solution chosen in Aos is to define a new heap block that prepends the additional system information to a normal record block. The block is formatted so that it meets the alignment requirements of a record block. This allows a non-active base object to be extended at a later time (perhaps in a different module) with an active body, and still remain compatible with its extensions. Type tests can be compiled in exactly the same way as before. The details of the new block are shown in appendix B.

Adding abstract root objects. The garbage collector finds all live data blocks in the heap by starting at the root pointers and exhaustively traversing all pointer links. All blocks that are not part of the transitive closure computed this way are unused and can be added to the free list for reallocation. For the traversal the location of root pointers and the location of pointers in the data blocks have to be known. This meta-information is contained in type descriptors, module descriptors and process descriptors. The type descriptors define where pointers are located in data blocks, the module descriptors define the location of global pointer variables, and the process descriptors contain the stack descriptors used to locate local pointer variables.

In Aos, the concern of locating root pointers is removed from the Heap module by using *abstract root objects.* Module descriptors and process descriptors declared in higher-level modules are extensions of this object. When the garbage collector finds such an object in the heap, it calls its FindRoots method to obtain root pointers contained in the object. In the case of a module descriptor this returns the global pointer variables and in the case of the process descriptor, the stack of the process is scanned for pointer candidates. This mechanism decouples the garbage collector from the module and process descriptor data structures, simplifying the implementation of other language environments on the kernel.

When the garbage collector is initiated, the root pointer of the module list is passed to it. This is sufficient to trace all reachable data structures as all other modules are reachable via the module list, and all processes are reachable via the scheduler data structures rooted in global pointer variables and active objects in the heap.

Finalization. *Finalization* is a mechanism that allows a user-defined procedure to be called when an object is deallocated by the garbage collector [55]. Finalization is not required often, but when it is required, it is essential [50]. It is useful to create state-reduced module interfaces, e.g., a file system module that does not require files to be closed explicitly.

Finalization is useful in cases where garbage collected objects manage external resources. It is then desirable to free the external resources associated with an object once the object is deallocated. As the garbage collector can not in general know the nature of the external resource, a suitably general mechanism must be provided.

Native Oberon supports finalization of objects using Templ's algorithm [120]. An object reference and associated finalization procedure can be registered with the garbage collector, which adds it to a special list of checked objects. When the garbage collector finds that a checked object is no longer reachable (except from the list), it removes it from the list, and calls the finalization procedure, passing a reference to the object. The object is not deallocated immediately, as a reference to it still exists. If the finalization procedure does not re-anchor the object, it will be collected in the next cycle.

In some cases a module needs to assign names to the objects that it manages (e.g., file objects in a file system module). When a client requests an object by name, the same object reference must be returned when the same name is requested multiple times. In Oberon this is implemented by linking the objects together in a global collection using *untraced pointers*, which are pointers that are ignored completely by the garbage collector. The objects are also registered for finalization. Subsequently, when a client no longer has a pointer to an object, the finalization procedure of the object gets called, and the module can remove it from the global collection.

Although this technique is simple to use and implement, it is subtle and if incorrectly used can lead to dangling references — errors that impact the stability of the whole system and are notoriously difficult to debug. It is therefore desirable to provide a finalization mechanism in Aos that does not depend on untraced pointers.

For the automatic management of named collections of objects the Aos kernel provides the *finalized collection* object type, which manages a collection of arbitrary objects with automatic removal of otherwise unreachable objects. Objects can be added to such a collection, removed from the collection and enumerated. They are also removed implicitly by the garbage collector when they are no longer reachable, except for the links inside the collection itself. Additionally, when an object is added, a *finalization procedure* can be associated with it, which gets called when the object is no longer reachable. Finalized collections are similar to identity directories in ETHOS [118], except that they can store arbitrary objects.

Unlike the Oberon system, the Aos garbage collector does not call the finalization procedure directly, as this can lead to deadlocks when the garbage collector is invoked while an application is holding some object locks. Instead, a separate process is used to call the procedures, thereby avoiding this kind of deadlock.

It is important to understand the limits of finalized collections when using them. As unreachable objects in a finalized collection are only discovered once a garbage collection cycle completes, it is possible that they remain in the collection for an indefinite amount of time. Truly scarce resources should rather be managed with explicit allocation and deallocation, even though it is less convenient and more error-prone.

Interaction with the page heap. The allocation algorithm in Aos is similar to that of Native Oberon, except in the case where it runs out of free memory. In Native Oberon a garbage collection is then performed, and the allocation retried once before failing. In Aos there is the additional choice of extending the heap by allocating more memory from the page heap shared with the process stacks (cf. 5.5).

Deciding whether to extend the heap or not is a difficult problem to solve in general. Due to the simplifying requirements that the heap memory be physically contiguous and that a non-moving garbage collector is used (cf. 4.3.1), once pages are allocated to the heap they can not be returned to the page heap. Therefore we would like to avoid growing the heap unnecessarily, to maximize the number of process stacks that can be allocated. On the other hand, if the heap is kept small, garbage collection has to be performed more often, resulting in unnecessary overhead. The compromise made is that the heap starts small, and while it is less than 50% of the total memory size, it is expanded immediately on initial allocation failure. Once it has grown past this limit, it is only expanded after attempting a garbage collection first. Once it has grown to 95% of the total memory size, it is not expanded any more. In this way the system adapts dynamically to its workload.

A simpler alternative to the dynamic sharing of pages between the heap and stacks would be to allocate a fixed number of pages to each role. This was rejected in favour of the more dynamic solution described above, but in some applications, e.g., embedded systems, it might arguably be the better choice.

Garbage collection algorithm. The mark-sweep garbage collection algorithm of Aos, adapted from Native Oberon, is described here.

The following auxiliary procedure starts at pointer p and marks all heap objects reachable from it that are not marked already. As a side effect, it adds any root objects discovered during the traversal to the global rootList.

```
PROCEDURE Mark(p: PTR);
```

Conservative marking of pointer candidates is handled by the two procedures below. The first procedure searches for pointer candidates in the specified block of memory using alignment checks and adds them to a global array. The second procedure sorts the array and scans the heap, matching candidates with actual heap block addresses and calling Mark in case of a confirmed candidate.

```
PROCEDURE RegisterCandidates(adr, size: LONGINT);
PROCEDURE CheckCandidates;
```

The following procedure implements the mark-and-sweep algorithm. Since the algorithm can not run concurrently with heap mutators, it is only executed on one processor, and all other processors are halted first. This is done by broadcasting an interprocessor interrupt to all processors. In response to this interrupt, one processor executes the garbage collector, while the other processors enter a waiting loop until it is finished. The interrupt handler pushes all registers on the stack, thus the conservative scanning of the stack will also discover pointer candidates that are contained exclusively in processor registers. The numCandidates variable counts the number of candidates in the global array. It is incremented by RegisterCandidates and cleared by CheckCandidates.

```
PROCEDURE CollectGarbage(root: PTR);
VAR obj: RootObject;
BEGIN
  rootList := NIL; Mark(root);
  REPEAT
    REPEAT
      WHILE rootList # NIL DO
        obj := Get(rootList);
        obj.FindRoots (* calls Mark and RegisterCandidates *)
      END;
      IF numCandidates # 0 THEN CheckCandidates END
    UNTIL (numCandidates = 0) & (rootList = NIL);
    CheckFinalizedObjects
  UNTIL rootList = NIL;
  Sweep
END CollectGarbage;
```

The CheckFinalizedObjects procedure checks the reachability of objects registered for finalization by traversing the list of registered objects once. If any unmarked objects are found, they are moved to another list and subsequently marked (including their subgraphs). A separate finalizer process is activated to call the associated finalization procedures after the garbage collector finishes. The Sweep procedure scans the heap and adds all unmarked objects to the free list. It also clears all mark bits.

The garbage collector is sensitive to memory caching issues, as it traverses all reachable objects in the heap. Since only one processor is active during garbage collection, the kernel makes sure that it always runs on the same processor. In some measurements this improved the performance of subsequent garbage collections with 35%, due to the primed memory caches, even when long periods of time with other program activity pass between the collections.

5.7 Interrupts and Exceptions

The IA-32 architecture supports three kinds of interrupts and exceptions [53]:

External interrupts These are signals from external I/O devices or interprocessor interrupts and timer interrupts from the APIC.

Software interrupts These interrupts are generated by special instructions under program control.

Processor exceptions These interrupts are caused by errors during instruction execution. They are subdivided into three categories: faults (restartable exceptions), traps (resumable exceptions) and aborts (fatal exceptions).

External interrupts can be masked by the processor using the disable interrupts instruction. I/O interrupts can also be masked individually by the interrupt controller. The other two kinds can not be masked.

Interrupts module. The Interrupts module is responsible for driving the interrupt controller and the low-level handling of interrupts using the procedure call model. Higher-level modules, e.g., device drivers using active objects, use the active object-based interrupt handling mechanism for I/O interrupts (cf. 4.3.3).

At this low level, all three kinds of interrupts described above are treated equally. Handler procedures can be installed in the interrupt vector table using the procedure defined below. A handler procedure is called when an interrupt occurs and gets full access to the processor state at that time. It can also modify the state on return, which is useful when implementing exception handling.

```
TYPE
  State = RECORD ... END; (* processor state *)
  Handler = PROCEDURE (VAR state: State);

PROCEDURE InstallHandler(h: Handler; int: INTEGER);
```

The state parameter contains a reference to the saved processor state at the time of the interrupt. When the handler returns, this state is restored in the processor. The stack pointer at the time of the interrupt is part of the state and can be modified to return on another stack. The instruction pointer is also part of the state and can be modified to return to another location, e.g., to the terminate procedure to terminate the process that caused an exception, or to a process-specific exception handler.

External interrupts are masked by the processor for the duration of the handler procedure. Specific external interrupts can be unmasked or masked permanently at the interrupt controller with the following two procedures. These are later used to implement the higher-level I/O interrupt handling.

```
PROCEDURE EnableInterrupt(int: INTEGER);
PROCEDURE DisableInterrupt(int: INTEGER);
```

During booting, every processor calls the following procedure to initialize its interrupt handling mechanism. It loads the address of the interrupt descriptor table into the interrupt descriptor table register. All processors share this table and handle interrupts the same way.

```
PROCEDURE InitInterrupts; (* initialize interrupt handling *)
```

Exceptions are treated exactly like interrupts, except that they can provide additional information on the processor state at the time of the exception, e.g., the kind of exception that occurred and the contents of debugging registers. The following procedure returns the additional information. The installed handler for an exception can modify the processor state, e.g., to cause the exception handler to return to a different location and unwind the stack (cf. 4.3.3).

```
TYPE
  ExceptionState = RECORD ... END; (* additional state *)

PROCEDURE GetExceptionState(VAR int: State; VAR exc: ExceptionState);
```

5.8 Modules, Types and Commands

Modules module. As in the Oberon system, the module called Modules is responsible for modules, types and commands. It declares the module descriptor and the list of loaded modules. It provides operations to load and unload modules and to find existing modules. A module can register a termination procedure to be called when it is unloaded and when the system is shut down.

For module loading the recursive algorithm of Templ [120] is used, with the small modification that the actual creation of the module in memory is performed by a plugin. This allows different object file formats to be supported (cf. 8.3.2), or for modules to be compiled on-the-fly. In configurations not requiring dynamic loading of modules (e.g., some embedded systems), the plugin need not be installed.

The types contained in modules are described by type descriptors, which describe the size of a type and the location of methods and pointer fields. These are used by generated code when calling methods and by the object allocator and garbage collector. Operations are provided to find type descriptors by name or by tag, which can be used for debugging and metaprogramming [120]. Type descriptors are generated by the Heap module, which also contains the garbage collector that uses the type descriptors to locate pointer fields.

Modules can contain commands, which are exported procedures that can be called dynamically without explicitly importing them (cf. 3.7).

Operations are provided to find commands by name. Aos commands are distinguished from Oberon commands in that they can take a generic parameter and return a generic result.

5.9 Processes and Synchronization

Active module. The Active module is described in detail in the following three sections. This module is responsible for everything directly related to active objects: processes (cf. 5.9), object locking (cf. 5.10) and condition management (cf. 5.11).

A *process* is used to execute the activity of an active object, expressed by the program declared in its body. The process is created when the active object is allocated, and terminates when the body exits. The process may call methods of the same or other objects and enter exclusive regions of those objects, where it competes with other processes for entry to these regions. An *object lock* is used to protect the exclusive regions of an object and allow only one process to enter a region at a time. *Condition management* concerns the implementation of the await statement and the evaluation of the conditions expressed therein.

5.9.1 Process States

A process can be in one of several states, and with every state (except *Terminated*) is associated a data structure which stores descriptors of processes in that state. It follows that a process can only be in one of these data structures at any time — when it switches state it is also moved from one data structure to another. The states and all possible transitions between them are shown in figure 5.6 and described below:

Ready This is the initial state of a process. It is ready to run and waiting for a processor to be assigned to it, at which time it will move to the *Running* state.

Running A process is in this state when it is running on one of the available processors. It leaves this state when it terminates, is preempted, or starts waiting for a lock, condition or interrupt.

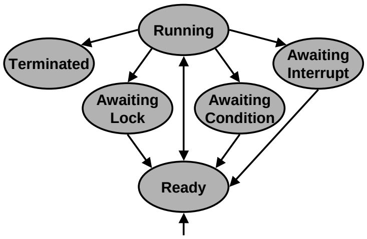


Figure 5.6: All process states and state transitions.

AwaitingLock A process is in this state when it is waiting to acquire an object lock that is held by another process. When it eventually acquires the lock, it moves back to the *Ready* state.

AwaitingCondition A process is in this state when it is waiting for a condition to be made true by some other process. When the condition is eventually found to be true by the runtime system, the process moves back to the *Ready* state.

AwaitingInterrupt A process (typically a device driver) is in this state when it is waiting for a specific interrupt to occur. This is similar to the previous state, except that the process will be enabled to run by an interrupt, and not the actions of another process (cf. 4.3.3). When it is enabled, it moves back to the *Ready* state.

Terminated This is the final state, which is entered when a process terminates. Terminated process descriptors are not in any program-defined data structure, but remain in the heap until they are cleaned up by the garbage collector.

5.9.2 Process Data Structures

This section describes the data structures used by the runtime system to keep track of processes: *ready queue*, *running array*, *lock queue*, *condition queue* and *interrupt array*.

Process Descriptor A *process descriptor* is a small record containing information about a process. Its most important fields are shown below.

```
TYPE
  Process = OBJECT ...
    stack: Stack;
    state: ProcessorState;
    preempted: BOOLEAN;
    condition: Condition;
    conditionFP: ADDRESS;
    priority: INTEGER;
    obj: OBJECT;
    next: Process
  END Process;

  ProcessQueue = RECORD
    head, tail: Process
  END;
```

Every process has its own private stack, referenced by the stack field, which is used to manage procedure calls and efficiently allocate and deallocate local variables.

When a process that was running on some processor is suspended, that processor's state is stored in the state field, so that it can later be restored on the same or another processor when the process is scheduled to run again. When the preempted field is set, the process has been preempted and the full processor state is stored in the state field. Otherwise, only a partial copy of the processor state is stored.

The condition and conditionFP fields used for object condition management are described in section 5.11.

The current priority of a process is defined by the priority integer field, which can assume one of a fixed number of values, with lower values indicating lower priorities.

Every process has exactly one associated active object, which is referenced by the obj field. This is set when the process is created and does not change afterwards. It is required only for the ActiveObject system call, which returns the associated active object.

A process descriptor can be in at most one queue, therefore a single next field is sufficient to implement queues of descriptors. A queue is represented by the ProcessQueue record, which references the first and last descriptor in the queue.

Ready Queue The descriptors of processes in the *Ready* state are stored in a global *ready queue.* In fact, a separate first-in-first-out queue is used for each process priority level (three queues are shown at the bottom of figure 5.7):

```
VAR
  ready: ARRAY NumPriorities OF ProcessQueue;
  maxReady: INTEGER;
```

A process descriptor is entered into the ready queue in constant time by indexing on its priority and adding it at the end of the relevant queue.

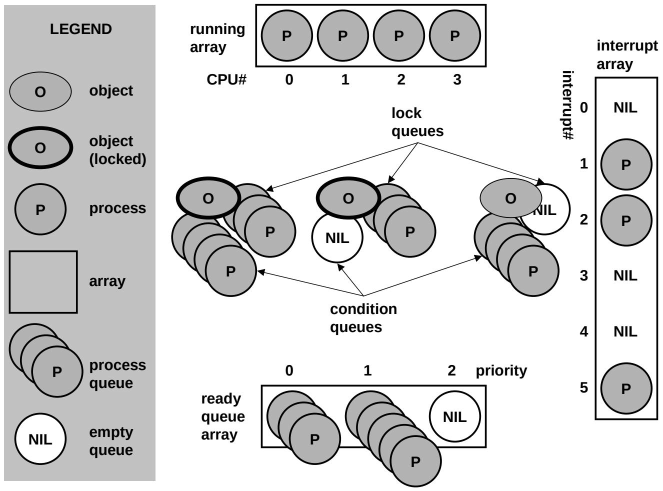


Figure 5.7: Overview of the process data structures.

Processes in the *Ready* state can not change their priority, so they are never moved from one queue to another.

To select a new process to run, the queues are scanned from the highest priority down until a non-empty queue is found and then the first process descriptor is removed from it. In the worst case all queues would have to be scanned, so, to reduce the average search time, the system keeps track of non-empty queues in the maxReady variable, which satisfies the following invariant:

$$\forall i : \text{MinPriority} \leq \text{maxReady} < i < \text{NumPriorities} : \text{Empty}(\text{ready}_i)$$

The full procedures to enter a process in the ready queue and select a process from the ready queue are shown below. The Get and Put procedures perform constant-time queue operations using the ProcessQueue type and Process.next field. It is assumed that the relevant spin lock is held when these procedures are called (cf. 5.4).

```
PROCEDURE Enter(p: Process);
BEGIN
  Put(ready[p.priority], p);
  IF p.priority > maxReady THEN maxReady := p.priority END
END Enter;

PROCEDURE Select(VAR new: Process; priority: INTEGER);
BEGIN
  LOOP
    IF maxReady < priority THEN new := NIL; EXIT END;
    Get(ready[maxReady], new);
    IF (new # NIL) OR (maxReady = MinPriority) THEN EXIT END;
    maxReady := maxReady-1
  END
END Select;
```

The Select procedure has a priority parameter to specify the minimum priority process that may be returned. This is used when the runtime system needs to select only process with the same or higher priority as the current process, e.g., when timeslicing.

Running Array While a process is running on processor i, its descriptor is referenced by element i of the global *running array* (four processor elements are shown at the top of figure 5.7). The runtime system indexes this array using the ProcessorID function, which returns a unique number for each processor it executes on.

```
VAR running: ARRAY NumProcessors OF Process;
```

The interplay between the running array and the ready queue is best illustrated by the following procedure, which can be called by a process to yield the processor voluntarily to another.

```
PROCEDURE Yield;
VAR id: INTEGER; new: Process;
BEGIN
  Acquire(Active);
  id := ProcessorID();
  Select(new, running[id].priority);
  IF new # NIL THEN
    Enter(running[id]);
    SwitchTo(running[id], new)
  ELSE
    Release(Active)
  END
END Yield;
```

The Acquire and Release operations on the Active spin lock are used to synchronize access to the shared data structures: the ready queue and running array. The Select and Enter procedures operate on the ready queue. The SwitchTo procedure performs a context switch by storing the processor state into its first parameter and loading the processor state from the second parameter (similar to Modula-2's TRANSFER). It does not return immediately, but rather 'returns' to the new process's last execution context. In all these procedures it is assumed that the caller holds the Active spin lock and the SwitchTo procedure releases the lock as it switches to the new process.

As the Yield procedure is running in the context of the current process and using its stack, it is important that the Enter and SwitchTo operations execute indivisibly. This avoids the following race condition which could occur if the spin lock were released between these two operations: Process P executes Yield (or another scheduling procedure) on one processor. After Enter, but before SwitchTo, process Q running on another processor also executes Yield, selects P to run and switches to it, whereby we end up with two processors executing P at the same time on the same stack! By holding the Active spin lock during the Enter and SwitchTo operations we avoid this problem.

Lock Queue A process enters the *AwaitingLock* state when it starts waiting for some object's lock to become available. Its process descriptor is added to the *lock queue* of the object being waited on. Every object with exclusive regions has such a lock queue, because processes can potentially wait to enter its exclusive regions. When a process releases the lock of some object, the runtime system removes the first process descriptor (if any) from the lock queue, cedes the lock to that process, and enters it in the ready queue (cf. 5.10). For illustration, figure 5.7 shows three objects, the leftmost two of which have three processes each waiting in their lock queues, and the rightmost one with no waiting processes. A process descriptor moves from the running array to an awaiting lock queue when the process attempts to lock an object that is already locked. As the process that has locked the object leaves the exclusive region, it hands the lock over to the first waiting process. When the process at the front of the queue receives the lock, it moves to the ready queue for its priority, from where it can proceed back to the running array.

Condition Queue The *AwaitingCondition* state is entered when a process starts waiting for some condition to become true in an exclusive region of an object. Its process descriptor is added to the *condition queue* associated with the object containing the await statement. Every object with exclusive regions containing await statements has such a condition queue. The queue is used by the runtime system when it needs to re-evaluate conditions associated with an object. If a condition is later found true, the associated process descriptor is removed from the condition queue and entered in the ready queue (cf. 5.11). For illustration, the middle object in figure 5.7 has no processes in its condition queue, and the other two objects have four waiting processes each. The movement of a process descriptor through the condition queue is similar to that through the lock queue, except that the queue can be rotated due to condition evaluation (cf. 5.11.3).

Interrupt Array The descriptor of a process in the *AwaitingInterrupt* state is stored in the global *interrupt process array*, which contains processes waiting on interrupts. When an interrupt occurs, the process is enabled and can then run to handle the interrupt (cf. 4.3.3). For illustration, figure 5.7 shows 6 interrupt numbers, with three processes waiting on interrupt number 1, 2 and 5, respectively.

No Process List A notable absence from the data structures described above is a global list of process descriptors. In fact, there is no such list that has to be updated when a process is created or terminates, which reduces space and time overhead. The described data structures are the only places where process descriptors are stored by the runtime system, so if its descriptor is no longer in any of these structures, a process is terminated by definition. Of course, all processes in the system can still be found by scanning the heap for their descriptors; in fact, this is exactly what a debugging utility that displays a list of active processes does. This may also find some descriptors of processes that have already terminated, but have not yet been garbage collected.

5.9.3 Process Creation

The runtime system creates a process with the following procedure. The body parameter specifies the body of the active object to be executed, priority is the initial process priority and obj is a reference to the associated active object.

```
PROCEDURE CreateProcess(body: ADDRESS; priority: INTEGER; obj: OBJECT);
VAR p: Process;
BEGIN
  NEW(p); NewStack(p, body, obj);
  p.preempted := FALSE; p.obj := obj; p.next := NIL;
  RegisterFinalizer(p, FinalizeProcess);
  Acquire(Active);
  IF priority # 0 THEN
    p.priority := priority
  ELSE (* inherit priority of creator *)
    p.priority := running[ProcessorID()].priority
  END;
  Enter(p);
  Release(Active)
```

```
END CreateProcess;
```

First, the process descriptor and stack are allocated and initialized. The stack and processor state in the descriptor are initialized so that the process will start executing at the beginning of the active object body and will call the Terminate procedure (cf. 5.9.5) when the body exits. For the process descriptor a finalizer procedure is registered to deallocate the process stack, which will be called by the garbage collector before it deallocates the descriptor.

Finally, the Active spin lock is acquired and the process descriptor is entered into the ready queue to be scheduled on some processor. The priority of the new process is set to the same as that of its creator, which has the consequence that the new process will never immediately preempt its creator due to a higher priority. Once the new process is scheduled to run, it can change its own priority.

5.9.4 Context Switches

The runtime system uses the SwitchTo procedure to perform a synchronous context switch from the running process to a new process. The version for the IA-32 architecture is shown below. The current processor state is stored in the running process descriptor and the new state is loaded from the new process descriptor.

```
PROCEDURE SwitchTo(VAR running: Process; new: Process);
BEGIN
  running.state.SP := SYSTEM.GETREG(SP);
  running.state.FP := SYSTEM.GETREG(FP);
  running := new;
  IF ~new.preempted THEN (* 1 *)
    SYSTEM.PUTREG(SP, new.state.FP);
    Release(Active);
    SYSTEM.PUTREG(FP, SYSTEM.GETREG(SP))
  ELSE (* 2 *)
    new.preempted := FALSE;
    SYSTEM.PUTREG(SP, new.state.SP);
    PushState(new);
    Release(Active);
    PopState
  END
END SwitchTo;
```

The caller of this procedure must hold the Active spin lock to synchronize access to the process descriptors and other shared data structures. Typically the descriptor of the running process has just been added to a scheduling queue, e.g., the ready queue or an object lock queue, and the descriptor of the new process has just been removed from such a queue. The latter descriptor is now stored in the running array (indirectly via the first parameter).

The next action of the procedure is to save the state of the running process's stack by storing the stack pointer and frame pointer in its process descriptor. No other registers are saved, as the IA-32 architecture has few registers and the calling convention does not use callee-saved registers.

Next there are two cases to consider (numbered 1 and 2 in the comments above):

1. The new process has previously performed a synchronous context switch by calling SwitchTo. Here we switch to the stack of the new process and then release the spin lock. It is safe to release the lock after switching the stack, as this leaves the old stack ready for reuse by another processor and the new stack in exclusive use by the current processor. All that remains is to switch the frame pointer to the new stack and then we can return in the context of the new process using the normal procedure exit code generated by the compiler.

2. The new process has previously been preempted asynchronously by an interrupt. Here also we switch to the new stack, but before releasing the lock we save the reference to the procedure descriptor on the new stack using the PushState procedure written in inline assembler. After the lock is released we should no longer access the old stack containing the procedure parameters, as it could already be in use again by another processor. The inline PopState assembler procedure is used to retrieve the new process descriptor and load the processor state from it, thereby directly returning to the context of the new process.

When suspending a process, the following help procedure is used to switch to a new process.

```
PROCEDURE SwitchToNew;
```

```
VAR new: Process;
BEGIN
  Select(new, MinPriority);
  SwitchTo(running[ProcessorID()], new)
END SwitchToNew;
```

The runtime system arranges that Select will always return at least one process, by creating a number of 'idle' processes that run at the minimum priority level reserved for them. One such process is created for every processor in the system.

Timeslicing. Timeslicing (cf. 4.2.3) is performed by calling the following procedure periodically from a timer interrupt handler to automatically preempt a long-running process. The state parameter contains the processor state that was active when the interrupt occurred.

```
PROCEDURE Timeslice(VAR state: ProcessorState);
VAR id: INTEGER; new: Process;
BEGIN
  Acquire(Active);
  id := ProcessorID();
  IF running[id].priority # Idle THEN
    Select(new, running[id].priority);
    IF new # NIL THEN
      running[id].preempted := TRUE;
      CopyState(state, running[id].state);
      Enter(running[id]);
      running[id] := new;
      IF new.preempted THEN
        new.preempted := FALSE;
        CopyState(new.state, state)
      ELSE
        SwitchToState(new, state)
      END
    END
  END;
  Release(Active)
END Timeslice;
```

First, the Active spin lock is acquired and a new process is selected to run. If no process is available, the current process is left alone and the procedure returns after releasing the lock.

If another process is available, the running process is preempted by copying the processor state at the time of the interrupt to the process's descriptor, setting the preempted flag in the descriptor and entering the descriptor in the ready queue.

Then the descriptor of the new process is stored in the running array and it is arranged to switch to the new process. If the new process has previously been preempted, the processor state is copied from the descriptor to the state parameter, from where it will be restored when the timer interrupt returns.

Otherwise, SwitchToState is called, which manipulates the state parameter so that the return will be to the location where the new process has previously called the SwitchTo procedure to perform its synchronous context switch.

Finally, the spin lock is released and the procedure returns, restoring the modified processor state and thereby returning to a different location from where the interrupt has occurred.

5.9.5 Process Termination

A process terminates itself by calling the following runtime procedure which simply selects another process to run.

```
PROCEDURE Terminate;
BEGIN
  Acquire(Active);
  SwitchToNew
END Terminate;
```

The process descriptor and stack are deallocated by the garbage collector (using the following procedure) once they are no longer reachable. Deallocating the stack is delayed like this for two reasons: First, it simplifies Terminate, because it does not need to deallocate the stack it is running on. Second, the stacks of terminated processes can be examined by a postmortem debugger.

```
PROCEDURE FinalizeProcess(p: Process);
BEGIN
  DisposeStack(p.stack)
END FinalizeProcess;
```

5.10 Object Locking

Object locks are used by the runtime system to implement the exclusive regions of objects. As exclusive regions are executed very often in an active object-based system, it is important to implement the locking mechanism efficiently.

5.10.1 Object Header

The runtime system adds a hidden header to every instance of an object type that has exclusive regions. This header contains fields used to implement object locking.

```
ObjectHeader = RECORD
  headerLock: BOOLEAN;
  lockedBy: Process;
  awaitingLock: ProcessQueue;
  awaitingCondition: ProcessQueue;
  ...
END;
```

The headerLock field is a spin lock to protect the other header fields from concurrent modification. The lockedBy field points to the descriptor of the process that is currently holding the object lock, and is NIL if the object lock is free. The awaitingLock queue contains descriptors of processes waiting on the lock and the awaitingCondition queue contains the descriptors of processes waiting on a condition to become true in the scope of the object.

5.10.2 Lock System Call

The compiler inserts a call to the Lock procedure at the start of every exclusive region. This procedure acquires the object lock if it is free, and otherwise suspends the running process.

```
PROCEDURE Lock(obj: OBJECT);
VAR r: Process;
BEGIN
  DisablePreemption;
  r := running[ProcessorID()];
  AcquireObject(obj.hdr.headerLock);
  IF obj.hdr.lockedBy = NIL THEN (* 1 *)
    obj.hdr.lockedBy := r;
    ReleaseObject(obj.hdr.headerLock);
    EnablePreemption
  ELSIF obj.hdr.lockedBy = r THEN (* 2 *)
    HALT("recursive lock")
  ELSE (* 3 *)
    Acquire(Active);
    Put(obj.hdr.awaitingLock, r);
    ReleaseObject(obj.hdr.headerLock);
    EnablePreemption;
    SwitchToNew
  END
END Lock;
```

Since the Lock procedure manipulates the running process's descriptor (referenced by variable r), it disables preemption during its execution to ensure that the running process does not change due to an interrupt.

Before modifying any fields of the header, the headerLock spin lock is acquired. This lock is only held for short times while the runtime system is modifying the object header and should not be confused with the actual object lock, which is implemented by the lockedBy field.

There are three cases to consider (numbered 1 to 3 in the comments above): the object lock is either free, held by the running process or held by another process.

1. If the lock is free, it is simply given to the running process (by setting lockedBy), the spin lock is released, preemption is enabled again, and the procedure exits. Measurements have shown that this is the common case, so it could be the first target for optimization.

2. If the object lock is held by the running process, this is an attempt to re-enter the exclusive region, which is not allowed.

3. If the lock is held by another process, the running process has to be suspended. First the Active spin lock protecting the scheduling data structures is acquired. Then the process descriptor is added to the object's awaitingLock queue, the object header spin lock is

released, preemption is enabled, and a new process is selected and context-switched to.

5.10.3 Unlock System Call

The compiler inserts a call to the *Unlock* procedure at the end of every exclusive region. This procedure releases the object lock and performs related actions.

```
PROCEDURE Unlock(obj: OBJECT);
VAR c: Process;
BEGIN
  IF obj.hdr.awaitingCondition.head = NIL THEN c := NIL
  ELSE c := FindCondition(obj.hdr.awaitingCondition)
  END;
  DisablePreemption;
  AcquireObject(obj.hdr.headerLock);
  IF c = NIL THEN
    Get(obj.hdr.awaitingLock, c);
    IF c = NIL THEN (* 1 *)
      obj.hdr.lockedBy := NIL
    ELSE (* 2 *)
      obj.hdr.lockedBy := c
    END
  ELSE (* 3 *)
    obj.hdr.lockedBy := c
  END;
  ReleaseObject(obj.hdr.headerLock);
  IF c # NIL THEN Acquire(Active); Enter(c); Release(Active) END;
  EnablePreemption
END Unlock;
```

The first action performed by the Unlock procedure is to check if any processes are waiting for conditions in the context of the current object. This is done because the running process is exiting an exclusive region and has possibly modified the state of the current object and caused some of the conditions being waited on to become true. If processes are waiting, FindCondition (cf. 5.11) is called to evaluate the waiting conditions. It returns the process descriptor of a process that can now continue to run because its condition has become true. If no such process is found, it returns NIL. At this stage the object is still locked, so the conditions are conceptually evaluated inside the exclusive region.

After the conditions have been checked, the lock can be released. As in the case of Lock, preemption is first disabled and the object header spin lock is acquired.

There are now three cases to consider (numbered 1 to 3 in the comments above):

1. If no condition was found true and no process is waiting for the lock, the lock can simply be released by setting lockedBy to NIL.

2. If no condition was found true, but there are processes waiting for the lock that is being released, the lock is granted to the process that has been waiting the longest (the first in the lock queue). As the object header spin lock is held at this stage, the lock is transferred atomically to the waiting process. There is no possibility for another process to execute the Lock procedure on another processor and 'jump the queue'.

3. If a condition was found true, the lock is granted to the process that had been waiting on it. As above, the lock is transferred atomically. There is no possibility for another process to enter the exclusive region in the meantime and falsify the condition again.

Next, the header spin lock is released. If a suspended process needs to be re-enabled because it was waiting on the lock or on a condition that is now true, the Active spin lock is acquired shortly and the process is entered into the ready queue. Then preemption is enabled again and the procedure exits.

5.11 Object Condition Management

The synchronization of active objects is based on the await primitive (cf. 4.2.3), which allows the programmer to specify arbitrary boolean synchronization conditions. The compiler and runtime system cooperate to evaluate the conditions and suspend and re-enable processes as necessary.

5.11.1 Await Statement

In Active Oberon, an AWAIT(b) statement, where *b* is an arbitrary boolean expression, is compiled into a helper procedure and an Await system call.

The helper procedure is equivalent to the following procedure, where the parameter specifies the frame pointer of the procedure containing the await statement and all references to local variables and object fields in the expression *b* are addressed relative to this parameter. A new name is generated for every helper procedure compiled and it can only be called by the runtime system.

```
PROCEDURE $Condition(fp: ADDRESS): BOOLEAN;
BEGIN
  RETURN "boolean expression from AWAIT statement"
END $Condition;
```

At the location of the await statement, the equivalent of the following code is compiled to perform the *Await* system call. The helper procedure ($Condition) and the current frame pointer (FP) are passed to the runtime system so that it can later re-evaluate the condition [31]. The self reference to the associated object instance is also passed as parameter.

```
IF NOT $Condition(FP) THEN
  Await($Condition, FP, SELF)
END
```

The generation of the inline IF statement by the compiler is an optimization. Measurements made while the system was hosting its own development on a dual-processor machine have shown that this avoids on average 70-85% of *Await* system calls. As a further optimization, the compiler could inline the condition in the IF statement, saving a procedure call at the cost of slightly more object code.

5.11.2 Await System Call

When the *Await* procedure is called, the IF statement generated by the compiler has already found the condition false, so *Await* has to release the object lock, suspend the running process and set up the runtime data structures so that the condition can be re-evaluated at a later time. Process descriptors are added to the condition queues of objects by the *Await* procedure and removed by the *Unlock* procedure. The Condition type matches the signature of the condition procedure generated by the compiler.

```
TYPE Condition = PROCEDURE (fp: ADDRESS): BOOLEAN;

PROCEDURE Await(condition: Condition; fp: ADDRESS; obj: OBJECT);
VAR r, t: Process;
BEGIN
  DisablePreemption;
  AcquireObject(obj.hdr.headerLock);
  Get(obj.hdr.awaitingLock, t);
  IF t = NIL THEN
    obj.hdr.lockedBy := NIL
  ELSE
    obj.hdr.lockedBy := t
  END;
  Acquire(Active);
  IF t # NIL THEN Enter(t) END;
  r := running[ProcessorID()];
  r.condition := condition; r.conditionFP := fp;
  Put(obj.hdr.awaitingCondition, r);
  ReleaseObject(obj.hdr.headerLock);
  EnablePreemption;
  SwitchToNew
END Await;
```

At the start of the *Await* procedure the object lock is released, so up to the point where the Active spin lock is acquired the implementation is similar to *Unlock*.

To suspend the running process, its descriptor is added to the awaitingCondition queue of the object. The condition procedure and frame pointer are saved in the process descriptor for later re-evaluation of the condition. Then the object header lock is released, preemption is enabled and a new process is switched to, as was done at the end of the *Lock* system call.

5.11.3 Evaluating Conditions

When the runtime system needs to evaluate conditions it calls the following procedure, which cycles through the descriptors in the condition queue and calls their condition procedures until it finds the first true condition. The relevant descriptor is then removed from the queue. If no true condition is found, the queue is left in the same state as it was before.

```
PROCEDURE FindCondition(VAR q: ProcessQueue): Process;
VAR f, c: Process;
BEGIN (* ~Empty(q) *)
  Get(q, f);
  IF f.condition(f.conditionFP) THEN
    RETURN f
  END;
  Put(q, f);
  WHILE q.head # f DO
    Get(q, c);
    IF c.condition(c.conditionFP) THEN
      RETURN c
    END;
    Put(q, c)
  END;
  RETURN NIL
END FindCondition;
```

To evaluate a condition, a normal procedure call is made through procedure variable condition stored in the process descriptor. As the condition procedure is compiled as a normal self-contained procedure, no full context switch has to be made. The frame pointer passed to the procedure gives it access to the local variables of the procedure containing the await statement and the fields of the relevant object. In a sense this is a *lightweight synchronous context switch* to the process that executed the await statement.

Measurements were made while the system was hosting its own development on a dual-processor machine to determine how often the first condition in the queue was immediately found true. When averaged over an extended period of editing and compiling, the first condition was found true about 90% of the times FindCondition was called. When the measurements were restricted to compiling only, an I/O-intensive activity implying many process interactions, the average dropped to just over 20%.

Interestingly, in all measured cases the average queue length when FindCondition was called was 1.00, which implies that if there are conditions to check when an exclusive region is exited, on average only one condition has to be checked.

To put the measurements in perspective, the condition queue was found empty in more than 98% of the cases where a process left an exclusive region. This means the FindCondition procedure was called in less than 2% of these cases, making the overhead of condition evaluation negligible in comparison with the locking overhead.

5.12 Handling Multiple Processors

The Processors module is responsible for multiprocessor support, viz., booting additional processors, interprocessor communication and local timers.

Booting other processors. At startup, the hardware elects a *boot processor* that starts running the firmware boot loader [51]. After the kernel has been loaded, this processor executes the module initialization code bottom-up until it gets to the Processors module, where the other processors are recognized and started.

The boot processor scans the configuration information supplied by firmware to find out at what address the APIC (cf. 5.1) is located and how many processors are available, as well as what their physical ID numbers are. Then it executes the following multiprocessor booting algorithm. For each processor to boot:

1. Set up a boot page in low memory with a simple 16-bit boot loader.

2. Use the APIC to send an *init* and *startup* interprocessor message to the specified processor, to initialize it and cause it to start executing at the boot page.

3. Wait with a timeout for the new processor to set a flag in shared memory showing that it has booted successfully.

When all processors have been booted, the boot processor enables global operations that influence all processors, e.g., garbage collection and TLB synchronization. Then it sets the flag signalling all other processors to start scheduling processes.

Each processor that boots up, executes the following:

1. The processor starts executing the boot loader on the boot page in 16-bit x86 mode.
2. The boot loader sets up a simple 32-bit execution environment.
3. The control registers, memory management, interrupt handling and APIC are initialized by Oberon code.
4. The processor adds its ID to the set of booted processors and sets the flag showing that it has booted successfully.
5. It waits on a shared memory flag signalling that all processors have been booted.
6. Finally, it calls the process scheduler to start executing processes (or an idle process if no others are available). The processor is now fully booted and taking part in normal active object scheduling.

Broadcasting messages. The Processors module provides a facility for asynchronously broadcasting an arbitrary message to all processors. This is implemented using an interprocessor interrupt and a message record in shared memory. All processors (optionally the sending processor also) are interrupted by the broadcast and act on the message before continuing where they were interrupted. This facility is used for:

TLB cache flushing When paging structures are changed, e.g., when a process stack is deallocated, the TLB cache on every processor has to be flushed, otherwise a processor might use stale information from its cache and corrupt memory.

Garbage collection At some stages during garbage collection, it is necessary to halt all processors except the one executing the collector. This is done by broadcasting a message to all processors. One processor reacts to the message by executing part of the garbage collection algorithm, while the others wait on a shared memory flag until the working processor signals that it has finished.

Memory cache control Each processor has its own caches and cache control registers. When a device driver needs to change caching behaviour, e.g., to enable write buffering on a frame buffer, this is done on all processors by broadcasting the relevant message.

The Broadcast procedure defined below is exported for use in device drivers. Its handler parameter specifies a procedure that will be called individually by all processors to process the message. The Message type can be extended to send arbitrary information with the broadcast. The constants are used as flags for the procedure. The Self flag specifies whether the calling processor should itself receive the message. The two barrier flags specify whether the processors should synchronize at the start and end of the message handling procedure, respectively.

```
CONST
  Self = 0; FrontBarrier = 1; BackBarrier = 2; (* flags for Broadcast. *)

TYPE
  Message = POINTER TO RECORD END;
  Handler = PROCEDURE (msg: Message; ...);

PROCEDURE Broadcast(h: Handler; msg: Message; flags: SET);
```

Local timers. The APIC on each processor has a timer which can be programmed to interrupt the processor periodically. On multiprocessor machines, this timer is programmed to perform timeslicing (cf. 5.9.4). The timer runs at the bus clock rate, therefore the bus clock speed has to be measured first, using an external timer device with a known frequency as reference. If the processor does not have an APIC (e.g., an older singleprocessor PC), the external timer device is used for timeslicing instead.

5.13 Kernel Services

Kernel module. The Kernel module is the main module of the Aos kernel. It provides facilities that require the support of all the other kernel modules. Unlike lower-level kernel modules, it does not export any implementation details and its interface can therefore be considered stable and portable, even when the kernel implementation changes. The interface consists of three sections: fine-grained timers, primitive atomic operations and finalized collections.

Fine-grained timers. The fine-grained timer facility is provided for accurate measurement of time periods. It is based on a counter that is periodically incremented at a fixed frequency. The GetTimer procedure below returns the current value of the counter. The timer frequency is a characteristic of the underlying implementation and is therefore exported as a variable, although its value remains constant.

```
VAR second: LONGINT; (* timer counts per second *)

PROCEDURE GetTimer(): LONGINT;
```

For more convenient access to the timer, some auxiliary procedures are provided to manipulate an abstract timer. The SetTimer procedure sets a timer to expire in the specified time, Expired tests whether a timer has expired, Elapsed returns the elapsed time and Left returns the time left before expiry.

```
TYPE
  Timer = RECORD END; (* opaque *)

PROCEDURE SetTimer (VAR t: Timer; ms: LONGINT);
PROCEDURE Expired (VAR t: Timer): BOOLEAN;
PROCEDURE Elapsed (VAR t: Timer): LONGINT;
PROCEDURE Left (VAR t: Timer): LONGINT;
```

Primitive atomic operations. In Active Oberon, any operation can be made atomic by wrapping it in an active object with exclusive methods, but for efficiency the kernel also exports some primitive atomic operations on simple data types:

```
PROCEDURE Inc(VAR x: LONGINT); (* x' := x + 1 *)
PROCEDURE Dec(VAR x: LONGINT); (* x' := x - 1 *)
PROCEDURE Add(VAR x: LONGINT; y: LONGINT); (* x' := x + y *)
PROCEDURE TestSet(VAR x: BOOLEAN): BOOLEAN; (* x' := true; return x *)
```

Finalized collections. The *finalized collections* mentioned in section 5.6 are provided with the interface shown below. The Add method adds an arbitrary object reference to a collection. If the fin procedure parameter is non-NIL, the specified procedure will be called before the object is deallocated. The Remove method removes an object from the collection. The Enumerate method enumerates the collection contents by calling the specified procedure once for each object currently in the collection (as long as the cont parameter is not set to FALSE).

Objects are automatically removed from the collection when they are no longer reachable (just before the optional finalization procedure is called). The finalization procedures are called from a separate process, so that they can acquire arbitrary locks.

```
TYPE
  Finalizer = PROCEDURE (obj: PTR);
  Enumerator = PROCEDURE (obj: PTR; VAR cont: BOOLEAN);
  FinalizedCollection = OBJECT
    PROCEDURE Add(obj: PTR; fin: Finalizer);
    PROCEDURE Remove(obj: PTR);
    PROCEDURE Enumerate(enum: Enumerator);
  END;
```

Chapter 6

System Services

This chapter describes the system services layer of the active object system (the middle layer in figure 3.1). The layer contains shared system services, e.g., the communication and file subsystems and their device drivers. Its modules can be classified into three kinds (cf. 3.6):

Service modules These modules provide standard interfaces for system services. They typically declare abstract objects that are implemented in plugin modules and used in client modules. A client locates a service by importing the service module and querying the registry exported there.

Plugin modules The plugin modules implement the interfaces defined by service modules, normally by extending the abstract objects declared there. Specific instances of the service objects are entered in the registry exported by the service module.

Helper modules These modules provide auxiliary functions, e.g., configuration and device driver support.

6.1 Files

Files are used for external storage. A file is a sequence of bytes normally accessed sequentially, but can also be accessed randomly. It can be named or anonymous. Files are normally used for permanent storage, but can also be used for temporary external storage (usually anonymous files).

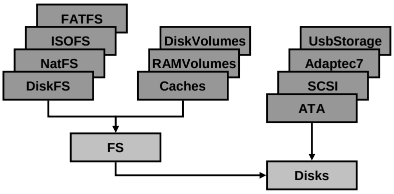


Figure 6.1: File subsystem structure.

6.1.1 File Systems

The Ceres Oberon system provided a single file system module with two compatible implementations. The first stored files permanently on disk, and the second stored files semi-permanently (until the next power-off) in RAM. In Native Oberon, there was a need to allow many kinds of file system for interoperability with other operating systems installed on the same machine, to access several kinds of removable media with their own file system formats, and to access file systems located on remote servers. The installable file system framework developed for Native Oberon was adapted and refined for Aos. An overview of the modules is shown in figure 6.1.

FS module. The FS (short for file system) module defines an abstract file system object that can be extended to define different kinds of file systems. An instance of such an object is responsible for managing each file system. As in the Disks module, concrete file system objects can be implemented in other modules that extend the types of the FS module.

The module defines a system-wide *file name space.* A file name consists of two parts: a file system specifier and a local file name, separated by a colon. The specifier is a prefix that uniquely identifies a specific file system currently accessible on the system. The local file name identifies a file uniquely in the specific file system (which might use a hierarchical naming scheme).

The data stored in a file system exists statically on disk or some other storage device. To make the data accessible to programs, a file system object has to be instantiated and registered in the file name space of the FS module. This is done automatically by the system (e.g., for the boot file system), or manually by a user with a file system command tool or configuration file. The process is called *mounting* as the file system is connected into the name space at this point. A unique prefix can be freely assigned to the file system at this time.

The FS module also defines an abstract volume object, which can be used as an intermediate object between a file system implementation and the underlying block device (a volume can be seen as a kind of virtual disk device). In this way, different file systems can be mapped to different partitions on the same disk, or a file system can be attached to a virtual disk in RAM.

Aos disk file system. The DiskFS module implements a concrete file system based on an abstract volume, of which two concrete implementations exist: module DiskVolumes implements a volume based on a disk device and RAMVolumes implements a virtual RAM volume. By combining a file system object with either of the two kinds of volume object, a file system on a real disk or a virtual disk can be accessed.

The implementation of the file system is based on that of Native Oberon, but the maximum file and volume size have been increased by increasing the volume block size and by using a double indirection in the file system structures. The disk volume object uses the Caches module to providing write-through and write-back caching, which significantly improves file system performance.

Caches module. The Caches module defines a generic cache object for block devices. A cache manages a fixed number of buffers that can contain data read from and written to different disks with the same block size. A write-through or write-back cache policy can be used. Clients of a cache call the Acquire and Release methods to obtain and return buffers, respectively. Clients should modify only the data field of a buffer — the rest of the buffer is under control of the cache object. The Synchronize method is used to write all modified buffers to disk.

```
TYPE
  Buffer = POINTER TO RECORD
    data: DataBlock; dev: Disks.Disk; block: LONGINT
  END;
  Cache = OBJECT
    VAR blockSize: LONGINT;
    PROCEDURE Acquire(dev: Disks.Disk; block: LONGINT;
        VAR buf: Buffer; VAR valid: BOOLEAN);
    PROCEDURE Release(buf: Buffer; modified, written: BOOLEAN);
    PROCEDURE Synchronize;
    PROCEDURE &Init(blockSize, cacheSize: LONGINT);
  END Cache;
```

The Acquire method acquires a buffer for the disk block specified by the dev and block parameters. If the buffer is already in the cache, it is locked and returned in the buf parameter with the valid parameter true, to indicate to the client that no disk read needs to be made (it is assumed that all clients access the disk through the cache). If the buffer is not in the cache, the cache returns the least-recently used unlocked non-dirty buffer for reuse. If all buffers are currently in use, it waits for a buffer to become available (it is assumed that all buffers are eventually released). The returned buffer is locked and returned with the valid parameter false (as it does not contain the desired disk block yet). The caller is responsible for reading the required block into the buffer.

The Release method releases a buffer to the cache. The modified parameter specifies whether the buffer has been modified by the caller since it was acquired and the written parameter specifies whether the caller has written the buffer to disk.

The Synchronize method writes all currently dirty buffers back to disk. This method can be called at any time to synchronize the buffer contents with the disk. The cache itself calls the method when non-dirty buffers become scarce.

The implementation of the cache object contains examples of non-trivial await statements. In one case the Acquire method waits for the state of a buffer object to change. Although the condition being waited on is apparently not local to the cache object (it depends on the state of a buffer, not the cache or method), it can only be established by one of the cache object methods modifying the internal state of the buffer. The state of the buffers can be seen as an extension of the cache object state, as it is only modified under control of the cache object in an exclusive region of the cache object.

In another case, the Acquire method has to wait for a non-dirty buffer to appear in the least-recently-used list. As it would not be efficient to traverse the whole list every time the condition is checked, the cache maintains a counter of the number of dirty buffers in the list. The counter invariant is established every time the list is modified.

The cache object manages disk blocks declared by a block device object, described below.

6.1.2 Disk Drivers

Disks module. The Disks module primarily serves to define the Disk object, which is an abstraction for general block devices. This object has some abstract methods that are extended and overwritten by device driver modules for hard disks, diskettes and other random-access block devices to implement concrete device driver objects. One such object is instantiated for each block device used in the system. A module that needs to access such a block device calls the methods of the specific instance directly. In this way the Disks module defines an *abstract interface* for block devices.

When designing the abstract interface, our main concern was to define a simple abstraction that is general, extensible, and does not impede performance. The essence of the interface is shown in the definition of the Disk object below. A disk is modeled as an array of equally-sized data blocks numbered sequentially. The two most important operations on it are modelled as methods. The Transfer method transfers (reads or writes) a contiguous collection of disk blocks between the disk and memory. The GetSize method returns the size (number of blocks) of the media currently in the drive. The block size is assumed to be fixed for a device, even if the media can be changed. It is exported as the blockSize field of the object.

```
CONST Read = 0; Write = 1; (* values for op parameter *)
TYPE
  Disk = OBJECT (Plugins.Plugin)
    VAR blockSize: LONGINT; ...
    PROCEDURE Transfer (op, block, num, ofs: LONGINT;
        VAR data: ARRAY OF CHAR; VAR res: LONGINT);
    PROCEDURE GetSize (VAR size, res: LONGINT);
  END Disk;
```

The Transfer method is synchronous (blocking), which means that it does not return until the transfer is complete, or an error has occurred (indicated by a non-zero res parameter). If parallel requests to a disk need to be made for performance reasons, this can be done by calling the method from different active objects. In this way all asynchronicity in the device interface is hidden in the device driver implementation, and not exported to higher levels with an asynchronous interface utilizing, e.g., callbacks or signals.

The Disk object is a plugin (cf. 3.6) and the Disks module provides a registry for disk objects. A module that needs to access a disk device queries the registry to find the required device by name. A disk device driver module instantiates a disk object for each device it manages and registers each instance here with a unique name.

```
VAR registry: Plugins.Registry;
```

The minimal interface presented above is sufficient for most current uses of block devices in the system. However, it is not possible to foresee all possible cases and define them in a standard interface. Therefore a generic message handler method is provided for message-based extension of the interface. The Handle method accepts a generic Message record, which can be extended in other modules when necessary. Handling of a message is optional; if it can not handled a non-zero result is returned. A few pre-defined messages (EjectMsg, GetGeometryMsg, etc.) are provided for common operations that are nonetheless not available on all devices.

```
TYPE
  Message = RECORD END;
```

```
Disk = OBJECT (Plugins.Plugin) ...
  PROCEDURE Handle (VAR msg: Message; VAR res: LONGINT); ...
END Disk;
EjectMsg = RECORD (Message) END;
GetGeometryMsg = RECORD (Message)
  cyls, hds, spt: LONGINT
END; ...
```

In addition to the functions described here, the Disks module also defines some auxiliary procedures for reading disk partition tables, which are used to partition a disk into different sections for different operating systems or file systems.

Disk device drivers. A disk device driver module extends the abstract disk object defined by the Disks module and implements at least the Transfer and GetSize methods defined there. When the module is loaded, it creates an instance of such an object for every compatible disk in the system, and registers the object with the plugin registry in the Disks module.

An example of such a driver module is the ATADisks module. It implements a driver for ATA (i.e., IDE) standard disks [8] connected to an ATA bus controller. The motherboard chipsets of most IA-32 machines have such integrated controllers, e.g., the Intel 82371SB [52]. Most of these controllers are backward-compatible with older models, but for the best performance, a driver has to be customized for a specific chipset. A possible approach is to make the concrete drivers extensible [75].

It is relatively simple to port Native Oberon disk drivers to Aos, as they use a similar interface to an abstract disk object. The main concerns when porting are protecting the objects using exclusive regions, adapting the interrupt handling (cf. 4.3.3) and adapting direct memory access to the Aos memory model (cf. 4.3.1). In this way Native Oberon drivers for Adaptec SCSI bus controllers [4] and USB storage devices [87, 123] have been ported by users of Aos.

6.2 Communication

The communication subsystem is an important component of the system, as it allows the system to access and provide services in the internet. It consists of three parts: the internet protocols, TCP agent services and link layer device drivers. An overview of the modules is shown in figure 6.2.

6.2.1 Internet Protocols

The basic internet protocols [54] were implemented for Aos, to allow it to support internet applications. The protocols were implemented in a modular fashion, with one module for every major protocol in the suite.

IP module. The IP module implements the closely-related *internet protocol* (IP) [92], *internet control message protocol* (ICMP) [91] and *address resolution protocol* (ARP) [88], which provide basic internet addressing and connectivity. The implementation is based on the abstract link device defined in the Net module, which can be instantiated with an ethernet driver for LAN connectivity or a *point-to-point protocol* (PPP) [113] implementation for point-to-point connectivity.

When IP datagrams are received, they are passed to higher-level modules for handling using the Input upcall procedure defined below. The Output procedure sends the specified IP datagram out on the specified link device.

```
TYPE
  Input = PROCEDURE (dev: Net.LinkDevice; VAR iphdr: Net.RecvHdr;
    hlen, dlen: LONGINT);
PROCEDURE Output(dev: Net.LinkDevice; VAR hdr: Net.RecvHdr;
    VAR data: ARRAY OF CHAR; hlen, dofs, dlen: LONGINT);
```

The ARP protocol is responsible for mapping IP addresses to link layer addresses on a LAN. Its main data structure is a table containing the current mappings. This is implemented as an object with exclusive methods for entering information in the table and a non-exclusive lookup method allowing concurrent lookup operations. The table also stores IP packets to be sent to addresses that are still being resolved. As soon as the relevant ARP reply is received, the packets are sent.

UDP module. The UDP module implements the *user datagram protocol* (UDP) [90], which provides simple connectionless datagram facilities to higher-level network protocols and applications. The implementation is based on the facilities provided by the IP module.

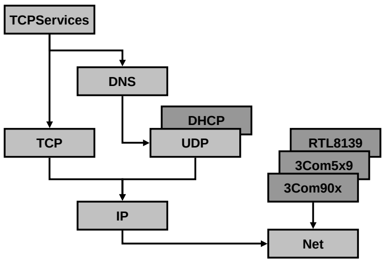


Figure 6.2: Communication subsystem structure.

The Socket object defined below functions as a UDP communication endpoint. A local UDP port number is associated with the object in the Open method. Datagrams that arrive for this port are buffered internally in the object until they are read with the Receive method. If called when no data is available, this method waits until data becomes available, or until the specified timeout expires. It returns the data of the datagram and the foreign IP address and port number. The Send method is used to send a datagram to the specified foreign IP address and port number. The Close method is used to stop buffering of incoming datagrams and to make the local port number available for re-use.

```
TYPE
  Socket = OBJECT
    PROCEDURE &Open(lport: LONGINT; VAR res: LONGINT);
    PROCEDURE Send(fip: IP.Adr; fport: LONGINT;
        VAR data: ARRAY OF CHAR; ofs, len: LONGINT; VAR res: LONGINT);
    PROCEDURE Receive(VAR data: ARRAY OF CHAR; ofs, size, ms: LONGINT;
        VAR fip: IP.Adr; VAR fport, len, res: LONGINT);
    PROCEDURE Close;
  END Socket;
```

DNS module. The DNS module implements a *domain name service* (DNS) [67, 68] client, which is based on UDP and mainly used to translate domain names (e.g., www.oberon.ethz.ch) into IP addresses (e.g., 129.132.178.197).

The module has a very simple interface. The HostByName procedure translates an internet host's domain name into an IP address. The HostByNumber procedure performs the reverse translation.

```
PROCEDURE HostByName(hostname: ARRAY OF CHAR;
    VAR adr: IP.Adr; VAR res: LONGINT);
PROCEDURE HostByNumber(adr: IP.Adr;
    VAR hostname: ARRAY OF CHAR; VAR res: LONGINT);
```

DHCP module. The DHCP module implements a *dynamic host configuration protocol* (DHCP) [32] client, which is used to initialize the TCP/IP subsystem without requiring manual intervention. The protocol is based on UDP and works by contacting a DHCP server on the

LAN during initialization. The server returns internet protocol parameters like the IP address, netmask, DNS server address, etc. IP addresses are assigned based on a static configuration for the host at the server, or are dynamically assigned by the server from a fixed pool of addresses. Automatic network configuration makes it possible to boot Aos from a CD or diskette and immediately use it to access the internet, without having to install it on the hard disk of a computer.

TCP module. The TCP module implements the *transmission control protocol* (TCP) [93], which is a connection-based protocol providing a bidirectional reliable byte stream connection between two internet hosts. The TCP protocol is significantly more complex than the other protocols presented so far. The Aos implementation is based on the ubiquitous BSD implementation of TCP [65, 135], but has been adapted to be more object-oriented and concurrent.

The Connection object defined below functions as a TCP communication endpoint.

```
TYPE
  Connection = OBJECT
    PROCEDURE Open(lport: LONGINT; fip: IP.Adr; fport: LONGINT;
        VAR res: LONGINT);
    PROCEDURE Send(VAR data: ARRAY OF CHAR; ofs, len: LONGINT;
        VAR res: LONGINT);
    PROCEDURE Receive(VAR data: ARRAY OF CHAR; ofs, size, min: LONGINT;
        VAR len, res: LONGINT);
    PROCEDURE AwaitState(good, bad: SET; ms: LONGINT; VAR res: LONGINT);
    PROCEDURE Close;
    PROCEDURE Accept (VAR client: Connection; VAR res: LONGINT); ...
  END Connection;
```

The Open method is used to open an active or passive connection, where an active connection is used by a client to contact a server, and a passive connection is used by a server to wait for client connections. The fip (foreign IP address) parameter is used to distinguish between the two cases, with a zero address specifying a passive connection. The lport parameter specifies the local TCP port number. If it is zero, a free port is assigned automatically. A connection is closed with the Close method, which performs a TCP half-close operation.

Opening and closing are asynchronous operations. The methods initiate the operation and return immediately, while the operation is performed in the background by an active object. The AwaitState method is provided to synchronize with the TCP connection state changes, e.g., to wait until a connection is established.

For data transfer, the Send and Receive methods are provided.

When a client connection request arrives on a passive connection, it is queued internally until it is accepted by the server calling the Accept method, which returns a new connection object for communication between the client and server. In the mean time, the passive connection can accept new client connection requests.

The connection object can be shared between multiple processes. This is typically used in servers to allow different agent processes to accept client connections on a shared passive connection. When a client request arrives, one of the agents will accept it and start communicating with the client. Meanwhile, the other agents continue waiting for client requests (cf. 6.2.2).

TCP implementation structure. Figure 6.3 shows the objects that take part in the Aos TCP implementation. The TCP module defines four types of object: *connection*, *connection pool*, *timer* and *sequence number source* (shaded in the figure). Two other types of object from other modules take part (unshaded in the figure): the *link device* (cf. 6.2.3) and *system timer*. One connection object is instantiated for every connection, a link device object is instantiated for every configured link device, and one instance of each of the other objects is shared between all the connections. The connection pool object stores all connections that currently exist, the sequence number source object generates initial send sequence numbers for connections and the timer object handles TCP timers.

The connection object encapsulates the state of a TCP connection and forms the core of the implementation. It reacts on messages from different sources, implemented as exclusive methods. Most of the methods are exported from the module and form the programming interface described above. The other methods are used internally for communication with the other objects taking part in the implementation.

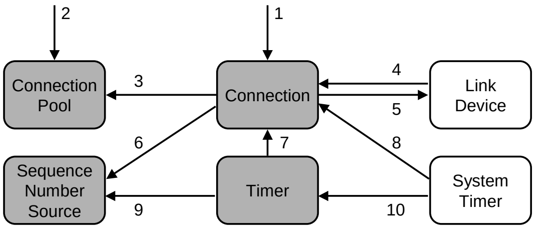


Figure 6.3: TCP object call graph.

Deadlock avoidance. In an environment where exclusive methods of one object calls exclusive methods of other objects, there is the danger of hold-and-wait deadlock. We now prove informally that this is not the case in the TCP implementation.

The arrows in figure 6.3 indicate which objects call methods of which other objects from their own methods. All exclusive method calls are shown, and no object calls its own exclusive methods. The method calls are:

1. The exported methods of the connection object, Open, Send, Receive, AwaitState, Close and Accept, can be called from outside the module, but only from higher-level modules, which are not called by the TCP module.
2. Likewise, the connection pool object exports Enumerate.
3. A connection object calls the SetLocalAddr, SetForeignAddr, SetAddrs and Remote methods of the connection pool object.
4. A link device object calls the Input method of a connection object when a packet arrives for the connection.
5. Conversely, a connection object calls the Send method of a link device object when it sends a packet.
6. A connection object calls the Get method of the sequence number source object when it is opened.
7. The timer object calls the DelayedAck and SlowTimer methods of a connection when the relevant timers expire.
8. The system timer object calls the HandleTimeout method of a connection when a timeout occurs.
9. The timer object calls the Update method of the sequence number source object periodically.
10. The system timer object calls the HandleTimeout method of the timer periodically.

The exclusive method call graph is acyclic by design (with one exception), and this can be verified informally by inspection of the source code. The exception is that connection and link device objects can call each other (cases 4 and 5 above). Therefore, this is the only case where a deadlock can possibly occur, as no cyclic waits are possible otherwise.

As an example of how deadlock can occur, consider the case where an acknowledgement packet arrives for a specific connection. The link device object calls the Input method of the connection (from an exclusive method) and the connection processes the packet, adjusts the send window and if buffered data is available, calls the Send method of the same link device to send out a data packet. If this method is also exclusive, it will result in deadlock.

The problem is solved in different ways in different link device drivers (cf. 6.2.3). For example, the 3C90X driver splits up the link device object into two objects internally and the 3C5X9 driver uses explicit locking and queues packets if the object is already locked.

Await statement examples. The connection object implementation contains some examples of non-trivial await statements. For example, the following condition in Send waits until the connection is established and the send window has space for a data segment, or until the connection is terminated (whichever comes first):

```
AWAIT(((state IN {Established, CloseWait}) & (sndspace >= len0)) OR
    ~(state IN {SynSent..CloseWait}))
```

With the following statement the Receive method waits until data is available, or the connection is terminated:

```
AWAIT((rcvhead.len # 0) OR ~(state IN {SynSent..Established, FinWait1, FinWait2}))
```

The following statement in AwaitState is used to wait for either a good or bad TCP state to occur, or a timeout. The former condition is typically signalled by packet processing and the latter by the timer object:

```
AWAIT((state IN (good+bad)) OR (Timeout IN flags))
```

The Accept method consists mainly of the following statement, which waits until the passive connection is closed, or a new client connection request has been queued by packet processing.

```
AWAIT((state # Listen) OR (acceptNext # NIL))
```

6.2.2 TCP Agent Services

The TCPServices module (definition below) facilitates the implementation of active object-based TCP server applications. The Service object defined in the module automatically handles connection acceptance and creates an active object (agent) to service each incoming client connection. The active object receives a handle on the connection with which it can receive client requests and send server responses.

```
TYPE
  NewAgent = PROCEDURE (c: TCP.Connection; s: Service): Agent;
  Service = OBJECT
    PROCEDURE &Start(port: LONGINT; new: NewAgent; VAR res: LONGINT);
  END Service;
  Agent = OBJECT
    VAR client: TCP.Connection;
    PROCEDURE &Start(c: TCP.Connection; s: Service);
    PROCEDURE Terminate;
  END Agent;
PROCEDURE OpenService(VAR service: Service; port: LONGINT; new: NewAgent);
PROCEDURE CloseService(VAR service: Service);
```

The Agent object is an abstract agent object, which is extended with an active body by a module implementing an actual agent. The exported client field is a handle on the TCP connection to the client. When the agent has finished processing client requests, it calls the Terminate method to signal to its service object that it is terminating.

A new service is instantiated with the OpenService procedure, where the port parameter specifies the TCP port on which the server should listen for client connections. The new procedure parameter specifies a generator procedure for agents that will provide the actual service. Every time a client connection arrives, the service object calls this procedure to generate a new agent to serve the client.

Example service: Echo. The example agent below implements the TCP echo service [89]. It receives all input from the client connection into a buffer and sends it back unchanged to the client. This continues until the client closes the connection ($\mathsf{res} \neq 0$) and then the agent terminates.

```
TYPE
  EchoAgent = OBJECT (TCPServices.Agent)
    VAR len, res: LONGINT; buf: ARRAY 4096 OF CHAR;
  BEGIN {ACTIVE}
    LOOP
      client.Receive(buf, 0, LEN(buf), 1, len, res);
      IF res # 0 THEN EXIT END;
      client.Send(buf, 0, len, res);
      IF res # 0 THEN EXIT END
    END;
    Terminate
  END EchoAgent;
```

The following procedure is used as a generator for echo agents. It simply allocates the relevant agent object, passing the supplied TCP connection as parameter to its initializer, and then returns the new object.

```
PROCEDURE NewEchoAgent(c: TCP.Connection;
    s: TCPServices.Service): TCPServices.Agent;
VAR a: EchoAgent;
BEGIN
  NEW(a, c, s); RETURN a
END NewEchoAgent;
```

To complete the example, the initialization of the service is shown below. The second parameter of the OpenService call specifies the TCP port number (the standard echo port is 7), and the third parameter passes the echo agent generator to the echo service object.

```
VAR echo: TCPServices.Service;
BEGIN
  TCPServices.OpenService(echo, 7, NewEchoAgent)
END
```

Example service: Web server. The following example outlines a simple HTTP 1.0 [11] web server agent. It shows how an agent can assign an input and output buffer on the client connection, so that it can efficiently parse client data character-by-character (in the ParseRequest procedure) and buffer the response sent to the client.

```
TYPE
  HTTPAgent = OBJECT (TCPServices.Agent)
    VAR ...
  BEGIN {ACTIVE}
    IO.OpenReader(in, client.Receive);
    IO.OpenWriter(out, client.Send);
    ParseRequest(in, uri, host, method, res);
    IF res = Ok THEN
      LocateResource(uri, host, method, type, f, res);
      IF res = Ok THEN
        WriteHeader(out, type);
        IF f # NIL THEN WriteFile(out, f) END
      END
    END;
    IF res # Ok THEN WriteStatus(out, res) END;
    IO.Update(out);
    Terminate
  END HTTPAgent;
```

The initialization of the service object is similar to the echo example, except that the HTTPAgent is generated instead of the EchoAgent, and the standard HTTP port 80 is used in the OpenService call.

6.2.3 Network Drivers

Net module. The Net module defines an abstract network link layer device object. As in the Disks module, other modules extend the object to implement device driver objects. A link layer device, e.g., an ethernet network controller, can send and receive packets on the network. It forms the lowest-level software part of the networking system.

For sending packets, the link device provides the Send method, which accepts a destination address, packet type and payload. The payload is split into a protocol header and data part. The device driver combines the link layer header, protocol header and data into one packet, which reduces the number of copy operations that have to be performed on the data. In order to keep the interface relatively simple, a more general data gathering facility is not provided. Typically, the send operation enqueues the data and returns immediately.

```
TYPE
```

```
  LinkDevice = OBJECT (Plugins.Plugin)
    VAR local, broadcast: LinkAdr; mtu: LONGINT; ...
    PROCEDURE Send(dst: LinkAdr; VAR hdr, data: ARRAY OF CHAR;
        type, hlen, dofs, dlen: LONGINT); ...
  END LinkDevice;
```

The local field of the link device contains the local link layer address and the broadcast field contains the link layer broadcast address. The mtu field defines the largest packet that can be delivered by the link layer.

Receiving of packets is implemented with an upcall mechanism. A receiver procedure for a specific packet type can be installed with the device object. When a packet of this type arrives, the procedure is called by the device. The protocol header of the packet is passed to the upcall, and the rest of the data can be obtained with the RecvData method.

```
TYPE
  Receiver = PROCEDURE (dev: LinkDevice; VAR hdr: RecvHdr;
      len, type: LONGINT; src: LinkAdr);
  LinkDevice = OBJECT (Plugins.Plugin) ...
    PROCEDURE InstallRecv(type, hlen: LONGINT; h: Receiver);
    PROCEDURE RemoveRecv(type: LONGINT; h: Receiver);
    PROCEDURE GetRecv(type: LONGINT; VAR h: Receiver; VAR hlen: LONGINT);
    PROCEDURE RecvData(VAR data: ARRAY OF CHAR; ofs, size: LONGINT); ...
  END LinkDevice;
```

As in the Disks module, the Net module provides a registry for link device objects.

```
VAR registry: Plugins.Registry;
```

Link layer device drivers. The first link layer device driver implemented for Aos was a driver for the 3Com 3C5X9 ethernet controller [1], which is relatively simple and well-documented. When the driver module is loaded, the controller is detected, initialized and a device driver object is allocated and registered with the Net module. A reference to the driver object is obtained from here by the higher-level protocol modules. When a protocol wants to send a packet, it calls the Send method, which waits until buffer space is available on the controller, then transfers the packet directly to the controller's buffer, and returns while the controller sends the packet out on the wire. For receiving packets, the protocol registers a receiver procedure with the driver object. When a packet arrives, the controller interrupts the driver object, which copies the packet to memory and calls the receiver to handle the packet.

At a later stage, a driver for the 3Com 3C90X ethernet controller [2] for Native Oberon [115] was ported to Aos. This controller uses direct memory access (DMA) to transfer packets from and to memory. This driver works similarly to the one described above, but also manages special buffers for DMA. The data from the network is deposited straight into these buffers by the controller and can then be copied directly to the application by the protocol.

Most recently, a driver for the Realtek RTL 8139 ethernet controller [97], which also uses DMA, has been implemented by a user of Aos.

6.3 User Interface

A user interface for the Aos system is being developed in a separate project. Although the window manager developed in that project already allows different applications to share the display and input devices, it is not rich enough yet to allow day-to-day work to be done exclusively in that environment. In the meantime Oberon for Aos (cf. 7.1) is used as the primary user interface for development, but it has the disadvantage that it does not allow arbitrary concurrency, thereby restricting the user interaction capabilities of active objects. This section describes the underlying user interface device drivers. An overview of the modules is shown in figure 6.4.

6.3.1 Display Drivers

Displays module. The Displays module defines the Display abstract raster device object that can be extended by display driver modules. Its interface is purposely kept small but general enough to be usable with most common raster display devices. The idea is to provide a uniform device interface for a higher-level graphics library, e.g., the one defined in [77]. The abstract raster device consists of a two-dimensional array of pixels with an integer device coordinate system.

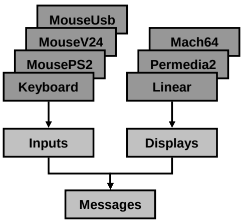


Figure 6.4: User interface subsystem structure.

Basic interface. The basic interface of the Display object is shown below. The Dot method draws a dot in the specified colour at position (x, y) and Fill fills the rectangular area specified by (x, y, w, h) with the specified colour. The Mask method fills the specified rectangular area with a tiled monochrome pattern in the specified foreground and background colour. The width and height fields export the size of the raster.

```
CONST
  red = 00FF0000H; green = 0000FF00H; blue = 000000FFH;
  trans = 80000000H; invert = 40000000H;
TYPE
  Display = OBJECT (Plugins.Plugin) ...
    VAR width, height: LONGINT; ...
    PROCEDURE Dot(col, x, y: LONGINT);
    PROCEDURE Fill(col, x, y, w, h: LONGINT);
    PROCEDURE Mask(VAR buf: ARRAY OF CHAR; bitofs, stride,
        fg, bg, x, y, w, h: LONGINT); ...
  END Display;
```

All these methods use an RGB true-colour model with colours encoded as 32-bit integers, usually written as hexadecimal constants. The low 24 bits specify the RGB components of the colour and the high 8 bits are used as flags, of which two are currently defined. The trans flag indicates a fully transparent colour and is used with the Mask method to 'paint' a pattern on the existing raster contents. The invert flag indicates that the colour should be combined with the existing raster contents in a reversible way — if the same colour is drawn twice, the original contents is restored — typically implemented with an XOR operation.

As in the Disks module, the Displays module provides a registry for display driver objects:

```
VAR registry: Plugins.Registry;
```

Low-level interface. For efficient block transfers to and from the raster frame buffer, the low-level interface below is provided. The Copy method copies the rectangular area specified by (sx, sy, w, h) to position (dx, dy). The Transfer method transfers a buffer containing colour values in the transfer format to, or from, the rectangular area specified by (x, y, w, h). The op parameter indicates the direction.

```
CONST
  get = 0; set = 1;
  index8 = 1; color565 = 2; color888 = 3; color8888 = 4;
TYPE
  Display = OBJECT (Plugins.Plugin)
    VAR format: LONGINT; ...
    PROCEDURE Copy(sx, sy, w, h, dx, dy: LONGINT);
    PROCEDURE Transfer(VAR buf: ARRAY OF CHAR; ofs, stride,
        x, y, w, h, op: LONGINT);
    PROCEDURE ColorToIndex(col: LONGINT): LONGINT;
    PROCEDURE IndexToColor(index: LONGINT): LONGINT; ...
  END Display;
```

Four transfer formats are defined, allowing a driver to expose a format that can be converted efficiently to its native frame buffer format. If a graphics library uses this low-level interface, it is responsible for mapping the driver's transfer format to its own internal format. The index8 format uses 8 bits to represent a pixel and the value is an index into a fixed colour table that can be queried with the ColorToIndex and IndexToColor methods. It is mostly intended for low-end devices with small memories. The color565 format uses 16 bits per pixel, with red, green and blue colour components encoded as 5, 6 and 5 bits, respectively. It provides high efficiency with a minimal true-colour model and is often used on medium-range systems. The color888 format uses 24 bits per pixel, with each colour component encoded as 8 bits. It is intended for memory-efficient true-colour devices. The color8888 format is similar to the previous one, but adds padding so that the frame buffer values are aligned on 32-bit boundaries, improving performance. This is the most common true-colour format.

Display device drivers. Similar to the Disks module, the idea is that a concrete display driver will extend the Display object and override its methods with device-specific implementations. However, the Display object is implemented in a way that simplifies writing a display driver significantly.

The Display object is not completely abstract, but contains a generic implementation of all the primitive methods in terms of a memory-mapped linear frame buffer. Therefore a minimal display driver implementation simply has to map its frame buffer into memory, initialize the format, width and height fields, and call the following method to initialize the generic implementation, where the parameters specify the location of the frame buffer.

```
TYPE
  Display = OBJECT (Plugins.Plugin) ...
    PROCEDURE InitFrameBuffer(adr, size: LONGINT); ...
  END Display;
```

The VESA linear frame buffer display driver of Aos takes advantage of the generic frame buffer implementation. VESA is an industry group that produced several display controller specification standards, which are implemented in the majority of display controllers on the market. Controllers supporting a linear frame buffer according to the VESA 2.0 standard [125] are usable with this generic driver, which provides full support for Aos, albeit without using any acceleration features of the relevant controller.

In the generic implementation the Dot, Fill, Mask and Copy methods are all implemented in terms of the Transfer method, and the Color-ToIndex and IndexToColor use a standard colour palette. Therefore a display driver module that is not based on a memory-mapped linear frame buffer can simply override the Transfer method and use the default, generic implementation of the others.

The generic S3 display driver of Aos is an example of such a driver. It overrides only the Transfer method, and re-implements it using the generic bank switching facilities available on S3 display controllers [104].

If the display controller has its own dedicated processor, which is the case with almost all modern controllers, better performance is obtained by utilizing this processor in the implementation of all the methods. The Aos driver for the 3Dlabs Permedia 2 [3] display driver is implemented in this way, based on the corresponding driver for Native Oberon [59].

6.3.2 Input Drivers

The Inputs module is responsible for user interface input devices and is based on the modules developed in [39]. It defines a general message-passing transport and some specific messages for different kinds of input devices. The message transport consists of the following definitions:

```
TYPE
  Message = RECORD END;
  Sink = OBJECT
    PROCEDURE Handle(VAR msg: Message);
  END Sink;
  Group = OBJECT
    PROCEDURE Register(s: Sink);
    PROCEDURE Unregister(s: Sink);
    PROCEDURE Handle(VAR msg: Message);
  END Group;
PROCEDURE NewBroadcaster(): Group;
```

The Message record represents a generic message that can be sent to interested parties, which are called *sinks* and are extensions of the Sink object. Typically, a message is broadcast to a group of sinks, which have registered themselves with a specific Group object. The group object is responsible for sending the message to all its registered sinks.

The Inputs module defines three types of messages for the three main kinds of user input devices: keyboard, mouse and pointer. A pointer differs from a mouse in that it provides absolute coordinates, whereas a mouse only provides information on relative movements.

```
CONST
  Release = 0;
  LeftShift = 1; RightShift = 2; LeftCtrl = 3; RightCtrl = 4;
  LeftAlt = 5; RightAlt = 6; LeftMeta = 7; RightMeta = 8;
TYPE
  KeyboardMsg = RECORD (Message)
    ch: CHAR; flags: SET; keysym: LONGINT
  END;
  MouseMsg = RECORD (Message)
    keys: SET; dx, dy, dz: LONGINT
  END;
  PointerMsg = RECORD (Message)
    keys: SET; x, y, z, mx, my, mz: LONGINT
  END;
VAR keyboard, mouse, pointer: Group;
```

A KeyboardMsg is sent whenever a key is pressed or released and the Release flag indicates which of the two cases occurred. The flags field also contains the current state of eight possible shift keys (LeftShift to RightMeta). The ch field contains the ASCII code of the key being pressed, or the NUL character if this is not defined. The keysym field contains an exact description of the key using the codes defined by the X Window system [105].

A MouseMsg is sent whenever a mouse movement or button state change is detected. The keys field specifies which mouse buttons are pressed, and the dx, dy and dz fields specify the relative movements in the relevant directions (the Z-coordinate is typically specified by an optional wheel on top of the mouse).

A PointerMsg is sent whenever a pointer device detects a change in position or button state. The keys field specifies which pointer buttons are pressed, and the x, y and z fields specify the absolute coordinates of the pointer. The mx, my and mz fields specify the maximum values of the relevant coordinates, and the minimum values are 0 in all cases. The origin is in the same relative position as defined in the Displays module.

The keyboard, mouse and pointer variables define global message broadcast groups for the three kinds of input devices. The default input devices are registered here when the system is configured, and the user interface subsystem registers itself here (cf. 7.1 and [40]).

Typically, a mouse is used as a virtual pointer device. For this purpose a virtual pointer object is defined which accepts mouse messages and uses them to update a virtual pointer position, which it uses to send pointer messages.

Based on the messages defined in the Inputs module, input device drivers have been implemented for all common serial mouse protocols and PS/2-style keyboards and mice [124].

Chapter 7

Application Case Studies

This chapter presents some applications that were developed on the Aos system to demonstrate how it can be used.

7.1 Oberon for Aos

A port of the ETH Oberon system was one of the first applications created for Aos. The main goal for this port was to allow ETH Oberon applications to run on Aos, specifically the Active Oberon compiler and development tools, which allows the Aos system to host its own development.

An additional requirement was that it should be possible to run Oberon as a full-screen application without the Aos window manager, but also with the window manager as a normal windowed Aos application. The rationale was that the full-screen configuration can be used on low-end machines that do not have sufficient resources to support the window manager.

The starting point for the port was the Native Oberon system. The implementation of nine low-level modules had to be modified or rewritten: Kernel, Modules, FileDir, Disks, Files, Display, Input, System and NetSystem. All other modules of the Native Oberon system and its applications, except for device drivers and some file system tools, could simply be recompiled, as the Active Oberon language is a superset of the Oberon language, and the module interfaces were kept compatible.

The functionality of most of the above nine modules is provided di-

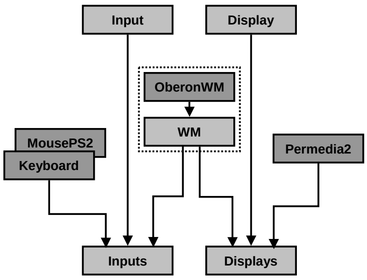


Figure 7.1: Oberon for Aos user interface modules.

rectly by the Aos system and their implementations therefore consist mainly of calling the equivalent Aos procedures. Some straightforward translation is necessary because of the object-oriented style of the Aos interfaces. However, the Display and Input modules were less straightforward to implement, due to the requirement that Oberon be usable with or without the window manager.

Oberon user interface. Figure 7.1 shows the low level modules concerned with the Oberon for Aos user interface. At the bottom are the two Aos modules Inputs and Displays (cf. 6.3.2 and 6.3.1). Above these are some concrete device driver modules for input and display devices (e.g., Keyboard, MousePS2 and Permedia2 in the figure). At the top are the Aos implementations of the Oberon modules Input and Display. In the middle are the modules of the Aos window manager (cf. 7.3) and a special driver called OberonWM (described below). These modules (boxed in the figure) are not required when running in full-screen mode.

The Oberon Input and Display modules only import the Aos Inputs and Displays modules, respectively. They are implemented in terms of the abstract input and display driver objects defined in the latter two modules. As such, the modules are completely independent of the window manager, and can function even when the window manager is not loaded. In this case, Oberon is in full control of the display and input devices.

When the Oberon system has to co-exist with the window manager, another solution is needed, as the window manager then needs to be in control of the display and input devices. In this case the OberonWM module is used to manage the Oberon window. It opens a window using the window manager and registers a virtual display driver with the Displays module and a virtual Oberon input driver with the Inputs module. The Oberon Input and Display modules then access the window indirectly via these virtual devices. Output from Oberon is sent to the window and keyboard input and pointer movements in the window are directed to Oberon.

Oberon and concurrency. As the Oberon system is based on a single-process cooperative multitasking model, many of its modules are not reentrant. This means that different processes must not call modules of the Oberon system in an uncontrolled way, otherwise data corruption can occur. Therefore, a single active object instance is used to execute the Oberon main loop in Aos.

Aos programmers are expected to distinguish between Oberon modules and pure Aos modules. Oberon modules are defined as modules that import the Oberon Kernel module (transitively), and all remaining modules are defined as pure Aos modules.

Pure Aos modules are required to export only reentrant interfaces. Oberon modules are not required to be reentrant, and should be used only from within the Oberon system. Of course, these requirements can not be checked automatically for general Oberon modules, and it is the programmer's responsibility to ensure that they are satisfied. The system merely ensures that only one instance of the Oberon main loop is started.

For experimental purposes, a global Oberon lock has been implemented, similar to the solution used in Concurrent Oberon [60]. This coarse-grained lock controls access to the complete Oberon environment and is acquired in the main loop and released for short periods of time when Oberon is idle. In this way active objects can safely be programmed to call Oberon procedures by surrounding all calls to Oberon modules with the relevant global lock and unlock operation. This solution is correct, but crude, as it was found to restrict concurrency significantly. A cleaner, preferred, solution when writing new modules is to use only pure Aos modules, with reentrant interfaces.

7.2 VNC Viewer

The Aos system with Oberon (cf. 7.1) can be used as its own development environment running on a personal computer. It provides various productivity applications: text editors, graphics editors, email clients, a web browser, file transfer applications, etc. However, in some cases an Aos user may still want to access applications running on standard operating systems like Unix and Windows, because a corresponding application is not available.

The *virtual network computing* (VNC) system [76, 100] provides an elegant solution to this problem. The *remote frame buffer (RFB) protocol* [101] defined by VNC allows an application and its user interface to be physically separated by a network. The idea is similar to the client-server separation in the X Window system [105], but the RFB protocol is more general and still lightweight and relatively simple to implement, especially on the client side.

Figure 7.2 shows how the VNC system can be used. Applications run on the server, using a virtual frame buffer for output. The virtual frame buffer is displayed on the client using the RFB protocol to communicate over the network. Likewise, user input on the client is sent to the application on the server. The system is more flexible than shown here and optionally allows multiple clients to connect to the same frame buffer, which enables application sharing. The server could also be a normal personal computer, which is shared by a local user and one or more remote users. The RFB protocol is stateless, so a connection can be closed and re-opened at any time, even from a different client, with the remote frame buffer still in the same state.

VNC clients and servers have been implemented for a multitude of systems, allowing application sharing between Unix, Windows, Macin-

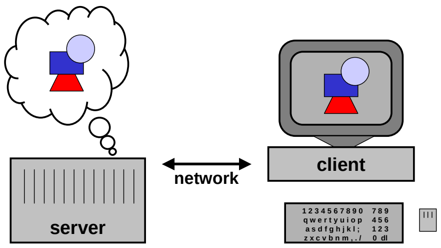


Figure 7.2: VNC example [100].

tosh, Windows CE, Palm and other systems [76]. Most of these implementations communicate via TCP/IP, but any reliable streaming protocol may be used.

To allow Aos users to use applications running on other operating systems, a VNC viewer (client) has been implemented, based on the VNC viewer for Oberon [58]. The viewer opens a window that displays the contents of the remote frame buffer (e.g., a Windows or Unix desktop) running under control of a VNC server on another (possibly shared) computer. Local mouse movements and keyboard events directed to the window are transmitted to the server, and control the applications running there. Remote display updates are transmitted by the server and displayed on the local machine.

The viewer implementation defines two types of active object: a receiver and a sender. The receiver receives display update events from the server and draws them in the window. The sender accepts local window events and sends mouse and keyboard events to the server. One instance of each of these objects is created for every VNC window opened. The implementation allows multiple VNC connections to be opened.

For normal non-multimedia applications the viewer performs well, even in a low-bandwidth LAN (10Mbps). It can be used over modem connections, but is not very responsive in this case. Multimedia applications like video playback require a high-bandwidth connection (100Mbps or more) for acceptable performance.

7.3 Other Applications

Except for the applications developed by the author and those ported from Native Oberon, several new applications have been developed by Aos users. These demonstrate that Aos is a stable development environment.

Parallel compiler P. Reali developed a parallel Active Oberon compiler [95] as part of his dissertation work [96]. The compiler is parallelized by instantiating an active object for the parsing and code generation of each scope in the source program.

Java environment P. Reali and R. Laich developed a Java environment with a just-in-time compiler [94, 102]. The Java runtime system is implemented in terms of the Aos kernel. For example, Java processes are mapped to active objects.

Window manager T. Frey is developing a window manager and text system based on a general display model [40]. It uses active objects to concurrently update the display.

VNC server T. Frey developed a VNC server (cf. 7.2) as plugin for the window manager. The server allows an Aos system to be remote-controlled via the network and to operate without a physical display.

Dynamic HTTP server T. Frey expanded the simple agent-based web server (cf. 6.2.2) into a complete HTTP 1.1 [36] server that can serve content dynamically generated by Active Oberon plugins.

FAT file system B. Egger developed a FAT file system for Native Oberon [35] and then ported it to Aos.

XML browser S. Walthert developed a general *extended markup language* (XML) toolkit and a XML/CSS browser [127]. The XML toolkit is used in the window manager and in the Aos configuration module.

Remote file system P. Stüdi developed a network file system [116] allowing Aos users to share files with a Unix system. The file system consists of a client running as an Aos file system plugin (cf. 6.1.1) and a server running on a Unix system. The client and server communicate via a proprietary stateless TCP/IP-based protocol that can recover from interruptions in the network.

Medical image processing D. Keller is developing a medical image processing server allowing Windows clients to store, process and retrieve medical images on an Aos server.

FTP server P. Reali and B. Egger developed an FTP server.

Chapter 8

Evaluation

In this chapter the Aos system is evaluated in four ways: section 8.1 makes a conceptual comparison with related operating systems, section 8.2 presents performance measurements, section 8.3 evaluates the portability and flexibility of the system and section 8.4 presents the sizes of the various subsystems.

8.1 Comparison with Related Systems

The flexible modular design of Aos makes it suitable for many different application areas and hardware environments, ranging from small embedded systems, through personal computers, to multiprocessor workstations and servers. Therefore, the system could be compared with a wide range of related operating systems. In this section, however, we restrict the comparison to a few shared-memory multiprocessor systems of historical significance, and a few Unix-related systems that are typically used on commercial multiprocessor machines.

8.1.1 Multiprocessor Research Operating Systems

Hydra

Hydra [136, 137] is the kernel of a multiprocessor operating system developed in the early seventies at Carnegie-Mellon University for the *C.mmp* shared memory multiprocessor. This machine was based on a number of PDP-11 minicomputers with a shared memory connected by a cross-bar switch.

The goal was to design a "collection of facilities of universal applicability and absolute reliability — a set of mechanisms from which an arbitrary set of operating system facilities can be conveniently, flexibly, efficiently and reliably constructed" [136]. This is the essential idea of a kernel, also strived for in Aos.

A good example of the Hydra design goal to *separate mechanism and policy* is its protection mechanism. An *object* is the unit of protection, and a *capability*, which is created and modified only by the kernel, references an object and defines which operations are valid on it. A call to a protected object is only possible via the kernel, which verifies that the capability allows the operation. The specific protection policy used is not part of the kernel.

Open extensible systems like Oberon and Aos allow parts of the system to be replaced or extended easily. Consequently there is less need to explicitly separate mechanism and policy, as required for comparable flexibility in a microkernel design.

Hydra *processes* communicate and synchronize with each other with simple message buffering primitives and Dijkstra-style semaphore operations. In Aos, processes communicate and synchronize by invoking active object methods and passing references to shared data directly to each other.

Firefly and Topaz

The *Firefly* [121] is a shared-memory multiprocessor workstation developed at DEC SRC in the mid-eighties. *Topaz*, the associated software system, provides binary emulation of the Ultrix (Unix) system call interface.

Like Topaz, Aos can also be used in a multiprocessor workstation environment. Unlike Topaz, it does not attempt any level of application compatibility with Unix, except for client-server cooperation via network protocols (cf. 6.2.2 and 7.2).

The Topaz system is structured as a microkernel. The kernel contains virtual memory management, thread scheduling, simple device drivers and the interprocess communication mechanism. The operating system, file system and window manager run as user space services.

The main advantages of a microkernel design are flexibility, protection and well-defined interfaces between services. These goals are addressed by other means in Aos. Flexibility is obtained with the use of modules and plugin objects. Hardware-based protection is deemed less important, because a type-safe language is used throughout, both for system and application programming. Well-defined interfaces are accentuated even more by modules, and, in contrast to most microkernels, are type-checked. There is no need for stub libraries to serialize and de-serialize parameters.

As is common in microkernel designs, the Topaz system separates the thread and address space concepts from the Unix (heavyweight) process concept. This distinction is made mainly to reduce the cost of creating processes (threads), so that server programs can service multiple requests in parallel. Communication between different address spaces and machines is performed by remote procedure calls. In Aos, creating an active object is similarly efficient, and servers can be programmed with multiple active objects (cf. 6.2.2).

The kernel and system are written in the Modula-2+ language, which extended Modula-2 with garbage collection, exception handling and concurrency. The applications perform reference counting (implemented by the compiler), and the garbage collector runs concurrently with them, performing an additional conservative mark-sweep operation to handle stack pointers and cyclic structures. A *Threads* module provides condition variables for synchronization and the language provides a *lock* statement for declaring critical regions.

Psyche

Psyche [108] is a parallel operating system that was developed at the University of Rochester in the late eighties for the BBN Butterfly Plus multiprocessor. The main emphasis was on experimenting with different parallel computing models co-existing on the same system, especially in the context of computer vision and robot control.

The Psyche kernel provides four abstractions: the *realm*, *protection domain*, *virtual processor* and *process*. A realm consists of code and data and is comparable to an object. Each protection domain has its own page tables, which map those realms that are accessible from the domain. Processes are threads of control implemented and scheduled at user level. Virtual processors are implemented by the kernel, and are used to schedule the user-level processes.

Every realm is at a globally unique virtual address, which allows processes to share pointers directly. An access to a realm not yet part of the current domain causes a page fault, allowing the kernel to check whether the access should be allowed. If an access is allowed, the realm is mapped into the invoking domain.

The bi-level scheduling allows different application-specific scheduling policies to be applied at user level and to co-exist. Software interrupts are used by the kernel to signal to a user-level scheduler implementation when scheduling decisions has to be made, e.g., when a process enters a new protection domain, or a timer expires.

In comparison, Aos provides only one kind of process and a single level of scheduling. It is not clear if the flexibility of different co-existing user-level scheduling policies in Psyche is really advantageous. If other scheduling policies are required in Aos, they can be implemented by modifying the Active module.

The Psyche kernel implementation uses four kinds of synchronization. Critical sections involving processor-local data structures are protected by disabling preemption of virtual processors, and those involving device handlers are protected by masking out interrupts. Other critical sections of small, bounded length are protected with spin-locks, which also disable preemption of virtual processors to comply with the length bounds. Critical sections involving conditions that are not bounded in time are implemented with (blocking) semaphores. Synchronization at user-level is under full control of the specific process scheduling implementation used.

The synchronization mechanisms used in the Aos kernel implementation are comparable to that of Psyche, except that semaphores are not used directly. Instead, active object exclusive blocks and the await statement are used for higher-level synchronization, also at the application ('user') level.

Hurricane

The University of Toronto's *Hurricane* [122] system was developed in the mid-nineties specifically to address the issue of scalability in large-scale multiprocessor operating systems and runs on the prototype Hector multiprocessor. It was argued that multiprocessor operating systems that evolve from singleprocessor systems only address concurrency issues by identifying and removing the most important bottlenecks, and do not address locality issues.

To address the scalability issue, a *hierarchical clustering* design is used, in which higher-level entities control resources in a coarse, global fashion, and lower-level entities control resources in a finer, localized way. Interaction between distant entities normally occurs at higher levels of the hierarchy. A *cluster* in the Hurricane system consists of a small-scale symmetric-multiprocessing microkernel sharing kernel data and control structures among a few processors. In a large system, several clusters are created to manage disjoint sets of 'neighbouring' processors, with replicated kernel structures. The clusters work together to present a consistent view of the whole system to applications.

Tornado

The University of Toronto's *Tornado* [42] system is the late nineties follow-up of the Hurricane system, also conceived to address the issues of locality and concurrency in shared-memory multiprocessor operating systems. It is implemented in C++ and runs on the Toronto NUMAchine and the SimOS simulator.

Tornado is structured in an object-oriented fashion and handles requests to different resources without accessing any common data structures or locks. A generalization of the Hurricane system's replicated kernel structures, called *clustered objects*, is used to partition a kernel object into different *representative objects*, which handle independent requests to a resource on different processors by different representative objects. The kernel sees to it that repeated requests to a service object by a client are serviced on the same processor as the client, and parallel requests are automatically served by different service threads.

The Tornado authors note that there are two kinds of locking in most systems: locks inherently related to concurrency control of shared data structures, and locks related to providing an *existence guarantee* that ensures that a data structure containing a variable is not deallocated during an update. They implemented a semi-automatic garbage collector for clustered objects, which ensures that a reference to such an object can be used at any time, even when no locks are held. They also claim that this removes a primary reason for holding locks across object invocations, thereby increasing modularity.

The Tornado argument for garbage collection also holds for Aos. In the presence of a garbage collector, it is never necessary to hold a lock on an object reference solely to protect the object from deallocation by another process while it is being used. Holding a reference is sufficient to ensure the existence of the object.

8.1.2 Unix and Related Operating Systems

Unix in all its flavours is the most widely-used server operating system on the Internet. Although the original system was developed for singleprocessors, Unix has been adapted for multiprocessor machines by various hardware vendors (e.g., Sun Solaris, SGI IRIX, IBM AIX, HP/UX) [16, 83, 106].

In this section we summarize implementation characteristics of some Unix-related operating systems, and make brief comparisons with Aos. Compared with the complexity of these large 'real world' systems, the transparency of Aos promises to be an advantage in building reliable, secure, and efficient custom applications.

BSD

As a representative example of a Unix system we look at the influential *BSD* ('Berkeley Software Distribution') system [65], which played a major part in popularizing the internet protocols.

Structurally, Aos is very different from BSD, which has a conventional monolithic kernel design, with separation between user and kernel (privileged) mode. This separation protects the kernel from accidental or malicious damage from applications, but also incurs overhead when an application performs a system call, since a cross-domain call is required, and often data has to be copied specially between user and kernel mode. In Aos, a kernel call is a normal procedure call to a kernel module.

Many facilities provided by the BSD kernel are also provided in Aos: memory management (in the Memory and Heap modules), interrupt and trap handling (in the Interrupts module), processes, context switching and time-slicing (in the Active module), and timing services (in the

Kernel module). However, in Aos these facilities are reduced in scope. For example, there are no separate address spaces and virtual memory (and consequently no explicit shared memory management and *mmap* system call), processes are lightweight (there is little process-connected state and no process groups), there is no user and group management (permissions, quotas) in the kernel, and no explicit interprocess communication (procedure calls are used instead).

Many other concepts and facilities from the BSD kernel have equivalents in the Aos system.

The interface of the Disks module can be compared with the *entry points* of BSD *block devices.* Common to both interfaces are *open*, *close* and *size* calls. Both interfaces also provide a function to initiate a data transfer: Transfer in Aos, and the *strategy* entry point in BSD. However, the latter always initiates an asynchronous data transfer, while Aos always uses synchronous transfers. This is done as they are deterministic and less error-prone to program, and experience in various microkernel-based systems have shown that synchronous transfers combined with lightweight processes are sufficiently powerful for most applications.

Like the BSD *block buffer cache*, the Cache object from the Caches module acts as an intermediary between the file system and block device driver, caching read and write requests. An important difference is that the cache object is optional, and an application that wants full control of a disk, e.g., a database system, can do so directly. In BSD, block device drivers must additionally implement the so-called *raw* interface for this case.

The FS module is comparable to the *file descriptor management* and *vnode layer* in BSD that manages file descriptors and provides an abstract interface to file system implementations. However, the Aos interface is not encumbered with a tight coupling between processes and file descriptors, and it focuses on persistent files, avoiding the unification of network 'sockets' and file descriptors. Additionally, multiplexing is handled by multiple active objects, obviating the need for additional polling, non-blocking or signalling interfaces.

Part of the reason why Unix processes carry a lot of state information is to simplify resource management. When a process terminates, the kernel can deallocate all the related resources, e.g., memory and file descriptors. In Aos, the system-wide garbage collector assumes this responsibility.

Solaris

Sun Microsystems' *Solaris* operating system, based on Unix System V Release 4 and BSD, supports symmetric multiprocessing [64]. It runs on singleprocessor machines and multiprocessors like the Sun Enterprise series with up to 64 processors. Here we focus briefly on processes and synchronization in the Solaris kernel. The rest of the kernel is comparable to the BSD kernel presented above.

Solaris uses a three-level process model, with traditional heavyweight Unix *processes* and two kinds of lightweight processes: *kernel threads* (also called LWP for lightweight processes) and user-level *threads*. The latter are managed by a user-level thread library and are scheduled by using a shared pool of kernel-level threads to execute them. Kernel threads can block during I/O operations (comparable to Psyche's virtual processors).

The Solaris kernel is preemptable and several kernel threads can be active in the kernel at the same time. It uses multiple synchronization mechanisms: *mutex locks*, *reader/writer locks*, *dispatcher locks*, *semaphores* and *condition variables*.

Several hundred lock instances are declared statically in the kernel sources, and several thousand locks can be created dynamically. The most common kind of lock used in the kernel is the *mutex lock*, which protects a simple critical section.

Two kinds of mutex lock are available: *spin locks* and *adaptive locks*. Spin locks operate as described in section 5.4. Adaptive locks are similar, with the following difference: when a thread attempts to acquire an adaptive lock that is held by another thread currently running on another processor, the first thread will spin waiting for the lock; otherwise, it will block. The reasoning is that it is likely that a lock held by a running thread will be released soon, as critical sections are normally short. Therefore it is not worth blocking the requesting thread and waking it up again later. Another difference between the two kinds of mutex lock is that adaptive locks may not be used in high level interrupt handlers, as they interact with the dispatcher (scheduler). Reportedly, the blocking case for adaptive locks rarely occurs.

When a mutex lock is released, and other threads are waiting to enter the critical section, Solaris 7 wakes up all the waiting threads at the same time. Earlier versions of Solaris only woke up one waiting thread, but this was found to complicate the code unnecessarily. Potentially the Solaris 7 behaviour can lead to the so-called *thundering herd* problem, where many woken threads waste processing time competing in vain for the lock, because only one can acquire it. Extensive testing has shown that this does not occur in practice, because critical regions protected with mutex locks are normally very short. Solaris does not transfer the lock directly to one of the waiting threads, because this would defeat the purpose of adaptive locks. If another thread attempts to acquire the lock before a woken thread is scheduled to run again, it can find the lock free and quickly enter and leave the short critical section. In case of a direct transfer, the new thread would have to block and be woken again later.

Reader/writer locks are used when there is a need to distinguish between read and write access to a shared data structure. At any time, several reading threads may be present in a critical section, or alternatively, one writing thread. These locks differ from mutex locks in their wakeup behaviour. If reader threads are waiting when a lock is released, all the waiting readers are woken, and if writers are waiting, only the first writer in the queue is woken. Additionally, the kernel does a direct transfer of the lock to the woken reader threads or writer thread. The reasoning is that this kind of lock is used for longer critical sections than the mutex locks.

The third kind of lock used in the Solaris kernel is the *dispatcher lock*. This lock has a simple implementation and is used to protected the scheduler and locking data structures. The first form of the lock is a normal spin lock implementation with no blocking option, and the second form additionally masks out low-level interrupts. High-level (realtime) interrupts are allowed to interrupt the scheduler, but their interrupt handlers are not allowed to invoke the scheduler. The dispatcher lock data structure is simply a byte flag indicating if the lock is held or not.

In addition to the three kinds of locks presented above, the kernel uses *integer semaphores*. The semaphore release operation is similar to the reader/writer lock release operation in that it wakes up only one waiting thread. Semaphores are only used in the buffer I/O module, the kernel module loader and a couple of device drivers.

The Solaris kernel uses *condition variables* in combination with mutex locks to perform process synchronization. Three operations are defined on condition variables: *wait*, *signal* and *broadcast*. The wait operation puts the calling process to sleep by removing its descriptor from the scheduler queue and adding it to a queue associated with the condition variable. The signal and broadcast operations remove processes from the condition variable queue and add them to the scheduler queue so they can run again. The signal operation removes one process, and the broadcast operation removes all processes that are waiting. The wait operation has some variations for different exception handling, priority control and time-out options.

It is interesting to study the evolution of the synchronization mechanism implementation in Solaris releases [64]. The kernel has benefitted from extensive testing over the years, and it is clear to see that the trend is towards more compact and efficient implementation of synchronization primitives like mutex locks.

Windows 2000

Microsoft's premier operating system, *Windows 2000* (formerly named Windows NT), supports symmetrical multiprocessing and runs on singleprocessor and multiprocessor Intel IA-32 machines [114].

Structurally, Windows 2000 is similar to Unix. It has a monolithic kernel that includes operating system services such as memory management, process management, I/O, interprocess communication, and also includes device drivers and the windowing, user-interface and graphics subsystems. Applications and some system services run as processes in user mode.

The kernel is divided into two layers: the *executive* and the *kernel proper*. The executive contains system service functions, some of which are exported to applications, and others that are only callable from within the kernel. The main services provided by the executive are: configuration manager, process and thread manager, security reference manager, plug-and-play device manager, power manager, instrumentation manager, cache manager, virtual memory manager, object manager, interprocess communication facility, run-time library functions and various executive support functions. The kernel proper provides fundamental services such as thread scheduling, synchronization and low-level hardware support.

For mutual exclusion inside critical sections, the kernel uses *spin* *locks*. A few core data structures of the kernel are protected with *queued spin locks*. These are spin locks with a waiting queue, which stores processors waiting for the lock. When a processor releases the lock, it hands the lock over to the next waiting processor. Queued spin locks do not access a shared cache line during busy waiting [103].

The Windows 2000 process model is akin to that of Solaris. Heavyweight *processes* act primarily as resource containers, and have separate address spaces. *Threads* are kernel-level lightweight processes that can share the address space of a process. Kernel processes and threads are complex objects — a process record has approximately 200 data fields, and a thread record approximately 180 (excluding sub-structures being pointed to). For comparison, the Aos process record (cf. 5.9.2) contains about 20 fields.

For thread synchronization, the kernel uses several kinds of so-called *dispatcher objects*: mutex, semaphore, queue, event, timer, process, thread and file. Each object has an associated signalling condition which changes according to the state changes of the object itself. For example, a queue object is signalled when an item is placed on the queue, and a timer object is signalled when a certain time interval has elapsed. A thread that executes a wait operation on a dispatcher object is suspended until the associated condition is signalled. Optionally, the thread can specify a maximum time it is prepared to wait, and it is possible to wait on multiple objects simultaneously. The approach taken by Windows is apparently to provide the most common synchronization scenarios pre-programmed as different kinds of dispatcher objects. In Aos, synchronization scenarios corresponding to those provided by Windows dispatcher objects can be programmed directly using the await primitive.

In Windows, as in Unix, kernel-mode code has complete access to system memory and device drivers run in kernel mode. This is not inherently a problem, but drivers are commonly written entirely in an unsafe language (C and assembler) and erroneous device drivers are the most common cause of system failure. Windows 2000 addresses this vulnerability using a driver certification mechanism. This mechanism warns the user when a third-party driver is installed that has not been authorized (and tested) by Microsoft, under the assumption that such drivers are less reliable [9].

Aos is also vulnerable to this problem, but at least drivers are largely implemented in a safe language, which excludes whole classes of possible errors. Furthermore, it is presumed that the vastly simpler kernel reduces the complexity of writing drivers, and consequently, the likelihood of errors.

8.2 Performance Measurements

The main reason for utilizing shared-memory multiprocessors is to improve application performance by parallel processing. Therefore, it is especially important for a multiprocessor operating system to perform well, otherwise the potential improvement through parallelization is wasted.

Although good performance is the ultimate goal of a multiprocessor operating system, providing a stable and reliable system should be the primary mission of any operating system developer, who must guard against unnecessarily complicating the design and implementation for the sake of performance. In short, performance is important, but should never come at the cost of dependability.

To see how Aos compares with an established operating system, *microbenchmarks* were used to compare selected parts of the Aos kernel with corresponding parts of a Unix kernel. The version of Unix used for the comparisons is Linux, which is widely deployed and highly optimized, due to its open source distributed development model.

A microbenchmark is a simple program that exercises a specific part of a system. Microbenchmarks are not always an ideal tool for comparing systems, because they are not realistic applications [12]. Nonetheless, their simplicity gives them the advantage that they can be used to compare specific system characteristics, when used carefully. Microbenchmark results can be useful even in isolation, since they provide an estimate of the performance of a specific part of the system, which can aid the development of efficient applications.

As Aos and Linux do not share common programming interfaces and models, approximately equivalent benchmark programs were developed in Active Oberon for Aos, and in C for Linux. Where active objects are used in the Aos benchmarks, Posix threads (not heavyweight processes) are used in the Linux benchmarks.

All performance measurements were run on a dual-boot Dell PowerEdge 8450 with 6 Pentium III Xeon processors running at 700MHz, each with 16+16KB L1 and 1MB L2 cache. The configuration had 2GB of RAM.

The Linux benchmark programs were compiled (with optimization) using the GNU C compiler and library version 2.95.2, and were run on Linux kernel version 2.4.1-SMP.

The benchmarking method is as follows. Each benchmark program executes the tested function repeatedly in a loop sufficiently long to measure accurate wall-clock time on an otherwise idle machine. The total elapsed time is divided by the number of iterations to obtain the time per function (the loop overhead was subtracted, where significant). Each experiment is repeated at least 5 times and the arithmetic mean, standard deviation and percentage error are computed for the results. The mean time approximates the cost of calling the function in an inner loop of an application program.

Minimal system call. When comparing operating system kernels it is customary to measure the *minimal system call* time, which gives an indication of the minimum overhead to invoke a system service. In Unix the system call that is usually measured is *getpid*, which returns the unique number of the current process. The equivalent in Aos is the ActiveObject function in the Active module, which returns a pointer to the active object executing on the current processor.

The results of the minimal system call benchmark (average of 5 runs of 10 million loops each) are:

μs/call		% error	
Aos	Linux	Aos	Linux
0.014	0.433	0.3	0.0

The result shows that the overhead of Aos for a simple system call is approximately 30 times less than that of Linux. This is mainly due to the fact that Aos is not split into user and kernel mode, and does not use hardware protection domains. The equivalent of a Linux system call in Aos is simply a statically bound procedure call.

Process creation. In the active object computing model, an active object is often created when an application needs to perform a new task. Therefore, an important performance characteristic of such a system is

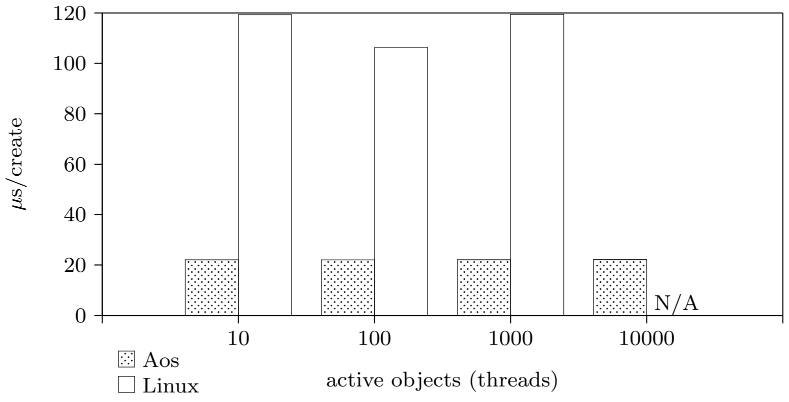


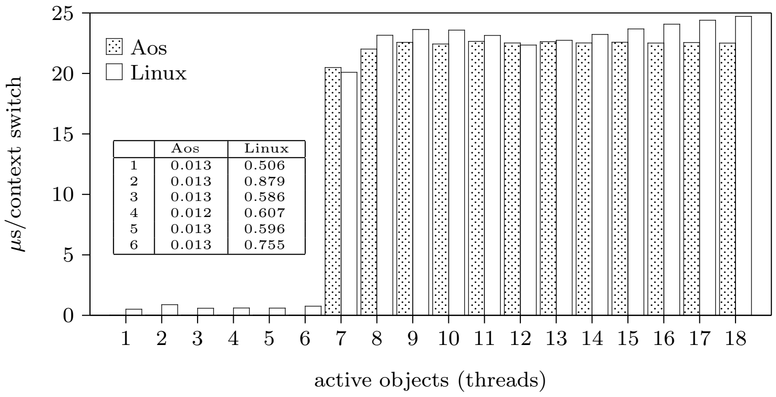
Figure 8.1: Aos active object and Linux thread creation time.

the time taken to create and destroy an active object. The equivalent in a conventional system is the time it takes to create and destroy a lightweight process.

The Aos version of the *process creation* benchmark defines an active object that does nothing except set a flag showing that it is terminating, and then terminates. It provides a Join method that can be called to wait until the termination flag is set. The main loop of the benchmark creates a number of such active objects, and then calls the Join method of each active object in turn to synchronize with its termination. The time measured includes this synchronization, the allocation of the active object, the invocation of its initializer, the creation of its process and the context switch to and from its process.

The Linux version of the benchmark uses the LinuxThreads implementation of Posix threads, included in the GNU C library. The benchmark creates a thread (with *pthread_create*), which does nothing but terminate itself (with *pthread_exit*). The main loop of the benchmark creates a number of such threads, and then calls *pthread_join* to synchronize with the termination of each thread. The time measured includes the *clone* system call to create the thread, the *exit* system call to terminate the thread and the *wait* system call to wait for the termination.

The results of the process creation benchmark (average of 50 runs)



	Aos	Linux
1	0.013	0.506
2	0.013	0.879
3	0.013	0.586
4	0.012	0.607
5	0.013	0.596
6	0.013	0.755

Figure 8.2: Aos active object and Linux thread scheduling time.

are shown in figure 8.1. The test was run with 10, 100, 1000 and 10000 simultaneous processes. The graph shows that the test time per process is independent of the number of processes already running. The percentage measurement error was less than 5% in all cases.

As Linux has a limit of approximately 1000 threads per address space, the test with 10000 simultaneous threads could not be run. In Aos the number of simultaneous active objects is limited by the configured maximum size of the process stacks. With the default 128KB maximum stack size per process, approximately 16000 active objects can be created. Therefore the Aos test could be run with more simultaneous processes than the Linux test.

For comparison, creating and terminating an active object in Aos is approximately 5 times faster than creating and terminating a thread in Linux.

Scheduling. The basic execution time overhead of processes are characterized by the *context switch time*, which is the minimum time required to switch execution from one process to another, including the cost of scheduling.

The Aos version of the *scheduling* benchmark defines an active object that loops calling the Yield procedure in the Active module, which releases the processor to another process that is ready to run. The main procedure of the benchmark creates a number of such active objects, and then pauses for a fixed number of seconds while the test objects run. Then it sets a global flag requesting the test objects to terminate, and synchronizes with their termination, similar to the process creation benchmark. Each test object counts the number of times it has looped, and from this the average time per context switch is computed.

The Linux version of the benchmark is similar, except that it creates Posix threads, and uses the *sched_yield* system call to release the processor.

The results of the scheduling benchmark (average of 5 runs) are shown in figure 8.2. The test was run with 1 to 18 simultaneous processes. Once again, the graph shows that the test time per process is independent of the number of processes already running. The percentage measurement error was less than 1%, except in the first 6 cases, where the Linux test showed a measurement error of up to 15%.

The measurements with up to six processes shows a special case of scheduling — when there are enough processors to service all ready processes (the test machine has six processors). The Aos and Linux schedulers test for this common case and avoid any additional work by directly returning to the current process. Consequently these cases are phenomenally faster than the rest — by a factor 1700 in Aos and factor 35 in Linux. The reason for the excellent performance of the special case in Aos is that it consists of a single test of an integer variable, which is pre-computed cheaply during normal operation. In Linux, the required parameter is not available directly, and has to be computed by looping over the per-processor data structures.

The results show that when the system is under light load (the exceptional case described above), the Aos scheduler is up to 60 times faster than Linux. When the system is under heavier load, the time per reschedule and context switch is roughly the same for Aos and Linux. The latter case could possibly be improved in Aos by splitting the scheduler critical section, thereby increasing the potential parallelism.

Locking. Using multiple processes can speed up an application, because it increases the potential parallelism. However, the application pays a price for this parallelism, in the form of locking overhead for critical sections in the manipulation of shared resources used by the

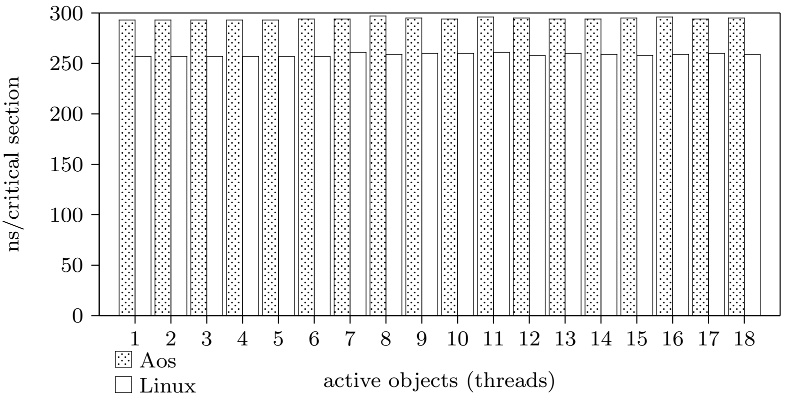


Figure 8.3: Aos exclusive region and Linux mutex lock time (independent critical sections).

processes when they interact. To design efficient parallel applications it is therefore important to know the cost of locking operations. The *locking* benchmark was designed to measure this cost, both in the case of no lock contention and in the case of heavy lock contention.

The Aos version of the locking benchmark defines an active object with an exclusive method that is called repeatedly from a loop in the body of the object. The main procedure of the benchmark creates several such objects that run simultaneously. In the first variation of the benchmark, each object calls its own exclusive method, independently of the other objects, and in the second variation, all objects call the exclusive method of the first object, which results in heavy contention of that object's lock. In both cases, the test objects terminate after a fixed number of iterations. The total time from when the first object is created until when the last one has terminated is measured and from this the average overhead per critical section is computed.

The Linux version of the benchmark is similar, except that it creates Posix threads, and uses *mutex locks* from the Posix thread library to perform the locking operations.

The results of the locking benchmark (average of 5 runs) in the case of no contention and heavy contention are shown in figures 8.3 and 8.4,

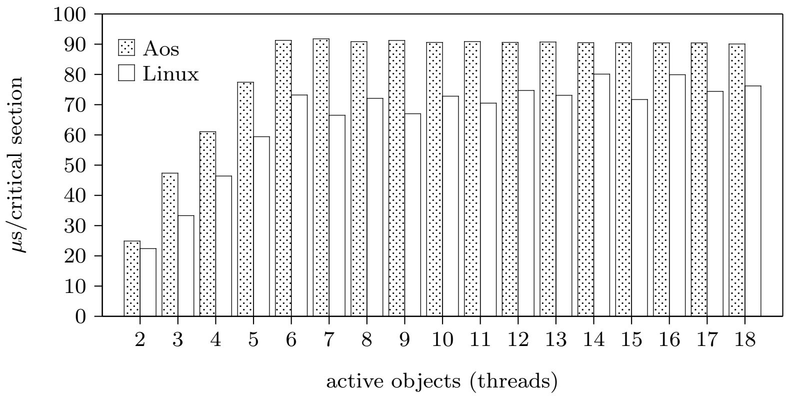


Figure 8.4: Aos exclusive region and Linux mutex lock time (with heavy contention).

respectively. The performance of Aos exclusive regions are somewhat slower, but still comparable with Linux pthread mutex locks. This could be attributed to the additional overhead of condition checking, but it is more likely due to the use of a fine-grained lock to protect the scheduler data structures. There are still opportunities to simplify and improve the performance of the locking implementation.

8.3 Portability and Flexibility

An appropriate way to evaluate the portability and flexibility of a system is to let someone other than the author adapt it to a different environment or purpose. The portability of the Aos system was evaluated by a port to the DNARD computer, summarized in section 8.3.1. Although the Aos kernel was designed to primarily support the Active Oberon language, it proved capable of supporting another language environment, as summarized in section 8.3.2.

8.3.1 DNARD port

B. Egger ported the Aos system to the DNARD (Shark) network computer [99] in an ETH diploma thesis project [34] supervised by the author. The full system was ported, including the active object runtime system, the Active Oberon compiler, the device driver and file system frameworks, the TCP/IP services, the window manager and XML browser, and the Oberon system and applications. Device drivers were developed for the display, keyboard, mouse and ethernet controller. The project was completed in four months of full-time work.

The Shark machine was chosen as a target because it uses an ARM processor [109]. These RISC processors are used in many mobile devices because of their low power requirements and relatively high performance. The port validated the applicability of the Aos design on a modestly-sized singleprocessor machine.

The work also serves as a guideline for porting Aos to other platforms, highlighting which parts of the system are less portable.

A major part of the porting effort was the development of an ARM backend with simple optimizations and an assembler for the parallel Active Oberon compiler (cf. 7.3). The compiler was first used as a cross-compiler running on a personal computer with Aos, and later ported to the Shark Aos system itself.

The adaption of the active object runtime system formed another major part of the work, of which memory management, interrupt handling and process management formed the largest subparts.

The development of device drivers was simplified by the PC-style architecture of the Shark, and the existence of a Native Oberon port [111]. As the Shark is disk-less, a simple hybrid RAM/TFTP file system was developed.

Once the compiler, runtime system and device drivers were done, porting the rest of the system was relatively simple, with only minor problems presented by some modules that use low-level language features (e.g., the SYSTEM module and inline assembler) for efficiency.

This project demonstrated that Aos can be ported to another architecture relatively easily. A possible improvement identified is to split the Active module into a processor-dependent and a processor-independent part. This would simplify future ports as only the processor-dependent part would need to be modified.

8.3.2 Java Environment

P. Reali, R. Laich and Gianni Banfi developed a Java virtual machine for Aos as part of a language interoperability research project [10, 96, 102].

The Java environment takes advantage of Aos kernel memory management, garbage collection, type support, method support and process support. A Java class library was ported, and its native methods were implemented in Active Oberon with the help of Aos system services.

The Aos module loader was generalized slightly to support module loader plugins. The Java environment uses such a plugin to integrate a byte code compiler that compiles Java class files on-the-fly. This allows Java classes to be loaded similarly to Active Oberon modules and enables bi-directional interoperability between Active Oberon and Java.

The Java language includes an interface concept, which allows a class to implement one or more pre-declared interfaces. A similar concept is not present in the standard version of the Active Oberon language and Aos runtime system. Therefore an additional runtime module was implemented to support interfaces, and the module loader was modified to automatically load this module when required. This mechanism was also used to develop an experimental version of the Active Oberon language that includes a similar interface concept.

The Java environment for Aos demonstrates the flexibility of the Aos core.

8.4 System Size

The sizes of the various Aos subsystems are shown in table 8.1. The middle three columns specify the size in bytes of the global variables, global constants (including strings), and the Intel IA-32 object code, respectively. The last column specifies the number of lines of (Active) Oberon source. More detailed figures are shown in appendix A.

For a native multiprocessor operating system, Aos is small, with a kernel of 7,210 lines of source or about 50KB of object code. For comparison, the 4.4BSD kernel (cf. 8.1.2) consists of 58,289 lines of C code (excluding file systems, network protocols and device drivers, which add another 143,962 lines) [65]. Version 2.4 of the Linux kernel consists of approximately 420,000 lines of C code (excluding drivers, file sys-

Subsystem	*Var*	*Const*	*Code*	*Lines*
Kernel	18088	1296	48434	7210
Service support	164	1620	30001	2532
File system	96	1928	55462	4624
User interface	128	792	20468	2204
Network	1512	3368	62126	6200
Oberon for Aos	2396	3332	112893	8667
Total	22384	12336	329384	31437

Table 8.1: Aos subsystem sizes.

tems and architecture-specific code, which bring the total to 2.4 million lines) [128], and has a minimum size of around 500KB on Intel processors. Microsoft boasts that Windows 2000 "consists of over 29 million lines of code", but does not say what is included in this figure, so it is not possible to compare specifics.

Chapter 9

Conclusion

This chapter summarizes the work that has been done and outlines possible future work. Specific contributions of this work are listed at the end of chapter 1.

9.1 Summary

Our goal was to develop an operating system as lean and transparent as Oberon, but more powerful and widely applicable, not only for single-user workstations, but also for servers, embedded control systems and small devices. The resulting system, called Aos, is based on the Active Oberon language which unifies objects and processes in the active object concept. The kernel of the system consists of a native runtime environment for Active Oberon.

The Active Oberon language is an extension of the Oberon language that allows the representation of concurrent processes as *active objects*, which are objects with intrinsic concurrent activity. For synchronizing processes, the unusual *await* primitive is used which allows the programmer to directly specify an arbitrary boolean precondition for continued execution of a process. Our contribution to the language development is a small, efficient and reliable multiprocessor runtime system running natively on modern machines.

The Aos operating system is a modular, extensible system in the style of Oberon, with modules arranged in three conceptual layers: the active object runtime system (kernel), shared system services, and applications. Except for the kernel, all modules are dynamically loadable, which allows flexible configuration and extension of the system at runtime. System service modules and their clients are decoupled with a simple *plugin module* technique, which facilitates interchangeable system services based on standard interfaces to increase flexibility. A more general implementation of the Oberon *command* concept is used to instantiate plugin modules and the objects under their control.

The primary purpose of the Aos kernel is to be a complete *runtime environment* for the Active Oberon language that also allows the programming of system-level services, e.g., device drivers. The kernel is tuned for this language, but has also been used to support the Java language. The object model supported by the kernel includes type extension (single-inheritance subclassing), type-bound procedures (virtual methods), exclusive type-bound procedures (synchronized methods), object-bound processes and general condition synchronization.

The first version of the Aos kernel was implemented for the Intel IA-32 *multiprocessor* system architecture, which has a typical small-scale (2–8 processor) symmetric multiprocessing architecture. The kernel is composed of several relatively small modules, each with a distinctive purpose and simple interface, which contributes to the transparency of the system. The areas covered by the various modules are (bottom-up): boot loader and hardware environment interface, fine-grained locks for kernel data structures, serial console output for debugging, virtual address space management, object storage management (heap and garbage collection), low-level interrupt handling, modules and types, active objects, multiple processors and portable kernel interface.

The middle layer of the system contains the shared operating system services (e.g., the file and communication subsystems) and their device drivers. Service modules typically export an abstract object-based interface, which is then implemented in a plugin module that is not directly imported by the client. This design helps to reduce the size of the system. The file subsystem defines a system-wide file name space, with names consisting of a file system specifier and a local file name. The specifier is a prefix used to locate a specific file system object, which then interprets the local file name part. This allows the name space to be extended at runtime with different kinds of file system implementations, e.g., the Aos disk file system that has a flat name space and the FAT file system that has a hierarchical name space. Disk devices are abstracted with a simple, but general, interface. The communication subsystem is responsible for communication with external machines. It defines and implements interfaces for the TCP/IP internet protocols based on abstract link layer device driver objects. The user interface subsystem provides basic support for user interfaces in the form of abstract display and input device objects.

The top layer of the system contains user applications. The most important example application developed by the author is a port of the Native Oberon system to Aos. This allows existing ETH Oberon applications to run on the new system, which also allows the new system to host its own development tools. The Oberon system can run as a full-screen application or in a window co-existing with other Aos applications. All in all, the implementation of only nine low-level modules of the Oberon system had to be modified and one additional driver module implemented. Another important application is the VNC viewer, which allows remote-controlling of applications running on other operating systems like Unix and Windows. Users of Aos have developed several other services and applications (including a parallel compiler, window manager, XML document-based user interface toolkit, Java environment, VNC server, dynamic HTTP server, XML browser, etc.), demonstrating the capabilities of the system.

The flexible modular design of the system makes it suitable for many different application areas and hardware environments, inviting comparison with a wide range of systems. The main advantage compared with large systems is its transparency, which promises to be an advantage when building reliable, secure, and efficient custom applications.

Specific characteristics of the Aos and Linux kernels were compared using microbenchmarks. In measurements of basic system call overhead and process creation overhead, Aos is 30 times and 5 times faster, respectively. These results show the reduced overhead of the extensible system approach. In measurements of scheduling and locking overhead, it compares favourably with the highly-optimized Linux threads implementation.

Users of the system have demonstrated its portability and flexibility in two projects. In the first, the system was ported to a different processor and system architecture in four months, including the development of a new compiler backend and device drivers. This is a significant development, because the relevant processor is used in many mobile devices, a possible application area for Aos. The second project demonstrated the flexibility of the runtime system through the development of a Java environment based on the kernel. This allows Java and Active Oberon programs to interoperate by sharing data and procedures.

9.2 Future Work

The aim of any operating system is to be a platform for further work. The various applications and extensions developed by users of Aos have demonstrated its applicability as a platform, but of course there is always room for improvement.

Conceptual improvements. An open question is how the Active Oberon language could be improved. In a parallel-running project [26], the type extension concept has been replaced with a definition concept, which enables object orientation based on interfaces instead of subclassing. A possible project is the reimplementation of the system using the new dialect, and a comparison of the size, complexity and performance of the two implementations.

A concept that was considered during the design of the system, but not realized in the end, is hierarchical namespaces for modules. It is still not conclusive whether a namespace facility is really needed, since the relatively small scale of the system allows simpler solutions. The solution taken was to name the modules of different subsystems or authors with short unique prefixes, thereby avoiding module name conflicts.

The cooperative environment assumption (cf. 3.1) was made consciously to simplify the system design and implementation. The strong typing and range checking built into the language avoids most problems caused by erroneous programs. However, if programs that use low-level facilities of the language or system have errors, they can cause the system to become unstable, e.g., due to memory corruption. One way to avoid this is to avoid using these low-level features. Conformance is easy to check by checking the import list of the modules concerned. Another solution might be to add a derivative of the *domain* concept of the Vamos system [84], allowing resources to be allocated on a domain-by-domain basis, and sections of the shared address space to be assigned to different programs, thereby containing faults to specific domains. But instead of complicating the system in this way, one might investigate where and why low-level features are used and replace them with safe alternatives, eventually removing all such unsafe features from the language.

Implementation improvements. The implementation of some portions of the system could probably be improved.

A minor point is that the size of the runtime system implementation can be reduced by removing some tracing and testing code. This would have the side effect of moving the Out module to a higher level of the module hierarchy, possibly to the system services layer, which would be conceptually cleaner.

Although the microbenchmarks of object locking have shown that its performance is comparable to that of Linux threads, it can probably be improved somewhat. A first step could be to simplify the implementation of locks even further, especially the interaction with fine-grained locks. There are also opportunities for tuning, e.g., inlining the most common locking operations, but it is questionable if this should be done: it would couple the runtime system and the compiler more closely, which would reduce flexibility.

The garbage collection algorithm (cf. 5.6) could be improved. The main problem is that it is a stop-and-collect algorithm that can not run concurrently with mutator processes. Although it is adequate for interactive and server use, for realtime use the delays in the order of tens of milliseconds may be too intrusive. Several possible improvements could be investigated [15, 50, 55]. The virtual memory manager could be utilized to perform a mostly-parallel collection, where the marking process runs concurrently with the mutator. Pseudo-incremental garbage collection could be used so that when the system is idle, a non-intrusive restartable garbage collection is initiated. The pointer rotation algorithm could be made non-intrusive (which probably only requires an additional tag field per heap block), so that it can run concurrently with mutator processes. Sweeping could be delayed until allocation time (lazy sweeping) to amortize sweep time and reduce the pause time of the collector. The cache performance of the algorithm might be improved by rearranging the heap structures. By using slightly more storage (e.g., one word extra per heap block) and not trying to save every bit, the algorithms could be simplified more and perhaps speeded up in the process.

Porting to other environments. A big strength of a small system is the ease with which it can be ported to new architectures and environments. A further advantage is that the type-safe language provides a clean abstraction for application programs and increases their portability also. Therefore a new port of the system can often immediately be used to run many existing applications without modification. This was demonstrated by the Shark port, which supported most existing applications once the basic system was finished. Ports to other ARM-based devices are under way, specifically a handheld device and an experimental wearable device.

Extended system services. The services provided by the system could be extended.

More standard network protocols and file systems could be implemented, as well as drivers for more devices. However, the amount of work to be done should not be underestimated. Standards are often developed by large corporations that have an entrenched interest in complexity, in order to frustrate competition [86]. The decision to implement Aos first for industry-standard hardware had the advantage that it could run on powerful and inexpensive machines, but also brought the burden of a wide variety of incompatible hardware components requiring custom drivers. For further development it would be advantageous to focus on niche applications where a small system has advantages, e.g., custom-built devices and embedded systems with a stable hardware composition.

The Java language environment could be developed further and other similar language environments implemented, but the same caveats as above hold for these environments.

Applications. Regardless of possible improvements, the system could be used in more applications, either for their own sake, or to further evaluate its strengths and weaknesses. A fruitful area might be the development of internet servers and network applications, perhaps in an embedded or mobile environment with small devices.

Appendix A

Module Sizes

Section 8.4 gives an overview of the sizes of various Aos subsystems. This appendix presents the module sizes in more detail.

The following tables show the sizes of the various modules that make up the Aos kernel and system. In each table, the first column specifies the module name. The middle three columns specify the size in bytes of the global variables, global constants (including strings), and the Intel IA-32 object code, respectively. The last column specifies the number of lines of (Active) Oberon source. The modules have all been compiled with the parallel Active Oberon compiler and include some debugging and testing code. The modules of the Aos kernel and system are prefixed with 'Aos' in the implementation, to prevent name clashes with existing Native Oberon modules.

Table A.1 shows the modules of the active object runtime kernel. These are defined as the minimum set of modules that can be taken together as a unit to support Active Oberon programs running on a bare IA-32 machine.

Table A.2 shows the modules that provide support for system services modules. They are not essential for the execution of Active Oberon programs and are therefore not considered part of the runtime system kernel.

Tables A.3 through A.5 show independent modules of the system services layer and table A.6 shows the base modules of Oberon for Aos.

Module	*Var*	*Const*	*Code*	*Lines*
AosBoot	2180	40	3462	808
AosLocks	4268	16	2227	428
AosOut	2048	96	3484	413
AosMemory	2916	24	6344	962
AosHeap	4236	424	9661	1127
AosInterrupts	1736	44	2930	609
AosModules	100	148	4698	503
AosActive	376	120	8900	1346
AosProcessors	220	280	5057	717
AosKernel	8	104	1671	297
Total	18088	1296	48434	7210

Table A.1: Runtime kernel module sizes.

Module	*Var*	*Const*	*Code*	*Lines*
AosPlugins	4	84	1999	176
AosCommands	4	148	2507	170
AosIO	60	92	7333	788
AosTrap	92	1004	8071	628
AosLoader	4	292	10091	770
Total	164	1620	30001	2532

Table A.2: Service support module sizes.

Module	*Var*	*Const*	*Code*	*Lines*
AosDisks	4	80	3455	310
AosATADisks	28	848	14166	1329
AosCaches	0	76	2457	220
AosFS	12	148	10058	1120
AosDiskVolumes	48	332	5266	337
AosRAMVolumes	4	56	1152	103
AosDiskFS	0	388	18908	1205
Total	96	1928	55462	4624

Table A.3: File system module sizes.

Module	*Var*	*Const*	*Code*	*Lines*
AosInputs	24	128	1846	298
AosKeyboard	40	80	5223	510
AosMousePS2	12	32	1472	169
AosDisplays	8	68	2485	267
AosDisplayPermedia2	40	336	8570	868
AosDisplayLinear	4	148	872	92
Total	128	792	20468	2204

Table A.4: User interface module sizes.

Module	*Var*	*Const*	*Code*	*Lines*
AosNet	36	76	1892	385
Aos3Com509	84	160	4756	455
Aos3Com90x	152	1260	8449	758
AosIP	184	336	5807	809
AosUDP	40	20	3239	317
AosDNS	120	104	5923	413
AosTCP	440	208	22289	2148
AosDHCP	0	600	3742	289
AosTCPServices	0	16	974	179
AosTestServer	20	76	1383	161
AosHTTPServer	436	512	3672	286
Total	1512	3368	62126	6200

Table A.5: Network module sizes.

Module	*Var*	*Const*	*Code*	*Lines*
Kernel	60	28	1214	422
Disks	0	16	744	158
FileDir	4	8	264	32
Files	12	44	5779	601
Modules	268	24	474	66
Objects	124	92	10276	882
AosOberonInput	4	120	1816	178
Display	1120	124	7217	666
Input	20	20	826	140
Viewers	28	24	4057	308
Fonts	16	76	4207	281
Texts	304	548	20485	1293
Oberon	120	252	8272	842
MenuViewers	4	64	5349	311
TextFrames	204	264	21888	1145
System	108	1628	20025	1342
Total	2396	3332	112893	8667

Table A.6: Oberon for Aos module sizes.

Appendix B

Technical Notes

This appendix describes the layout of heap blocks in detail. It is intended for reference when reading the Oberon source of the Heap module (cf. 5.6).

Object Storage Layout

Figures B.1 through B.6 show the layout of heap blocks. Every heap block has a tag word in front of it pointing to a type descriptor for the block. The layout is based on Native Oberon, except for the protected record block (figure B.6), which is unique to Aos. The type descriptors were slightly modified to allow pointer offsets to be negative. The Native Oberon structures are based on those of other ETH Oberon systems [27].

All heap blocks start with a tag, which is a pointer to a separate type descriptor block or a meta-type descriptor contained in the heap block itself. Bit 0 of the tag — the *mark bit* — is used during the mark phase of the collector to mark a block that has been reached by the traversal. It is cleared during the sweep phase, so that type tests can be performed without having to mask the low bits. Bit 1 — the *array bit* — is set on an ArrBlk and cleared on all other blocks. With the low two bits masked off, the tag value points to the size of the block, which is stored in the type descriptor or meta-type descriptor (the masked bits are indicated with an '`M`' in the figures). The field offsets of pointer values in the heap block are stored directly following the size value (see figure B.5), and are terminated by a sentinel that also encodes the number of pointer offset values. During the mark phase the tag (excluding the low two bits) is modified to step through the pointer offset table and thereby store the recursion state of the traversal [85]. When the garbage collector is not running, Bit 2 of the tag is always cleared, and Bit 3 is set if the block has a real type descriptor, and cleared if it has a meta-type descriptor.

The alignment of heap block fields are indicated on the right-hand side of the diagrams. The first data field of a heap block is always aligned so that any basic data type can be placed at that address (address divisible by 8). In some cases the garbage collector uses the alignment restrictions to recognize what kind of block a pointer is pointing to, by examining the low bits of the pointer value. These bits are indicated in the diagrams, with '`X`' characters indicating don't-care bits.

The SysBlk, TypeDesc and ProtRecBlk have a meta-type descriptor included at the start of the block. This serves only to describe the size of the block itself, and includes an empty pointer offset list. It is possible that the conservative marking algorithm used for process stacks finds a phantom pointer to the type descriptor, and care has to be taken to handle these cases in the garbage collector. Possible phantom pointers are indicated with dashed boxes in the figures.

In Native Oberon, a negative sentinel value is used to encode the number of pointer offsets in a type descriptor and meta-type descriptor. In Aos, this is not possible as a ProtRecBlk also has legitimate negative pointer offsets in the object header. Instead, a small negative constant (MPO), which is less than the smallest negative pointer offset, is used as sentinel. When the garbage collector finds a pointer offset less than this value, it has reached the end of the pointer offset list, and can retrieve the number of pointer offsets from the value found by subtracting MPO.

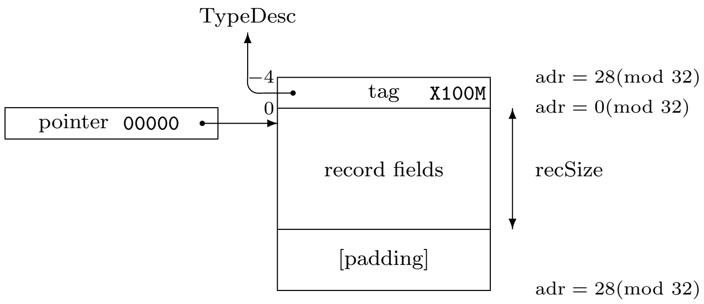


Figure B.1: A RecBlk is used for a record.

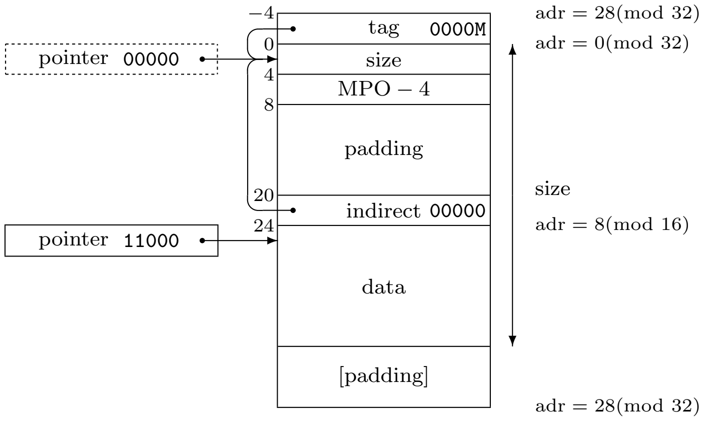


Figure B.2: A SysBlk is normally allocated with SYSTEM.NEW.

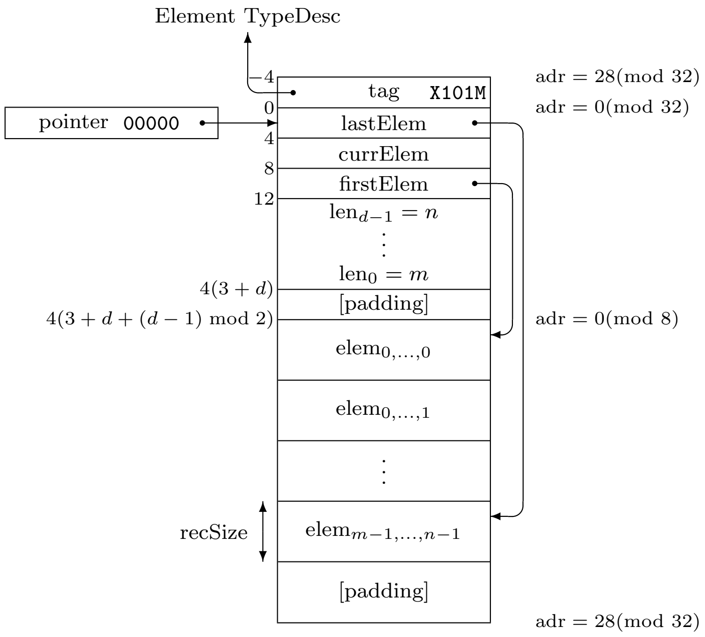


Figure B.3: An ArrBlk is used for an array when the elements contain pointers.

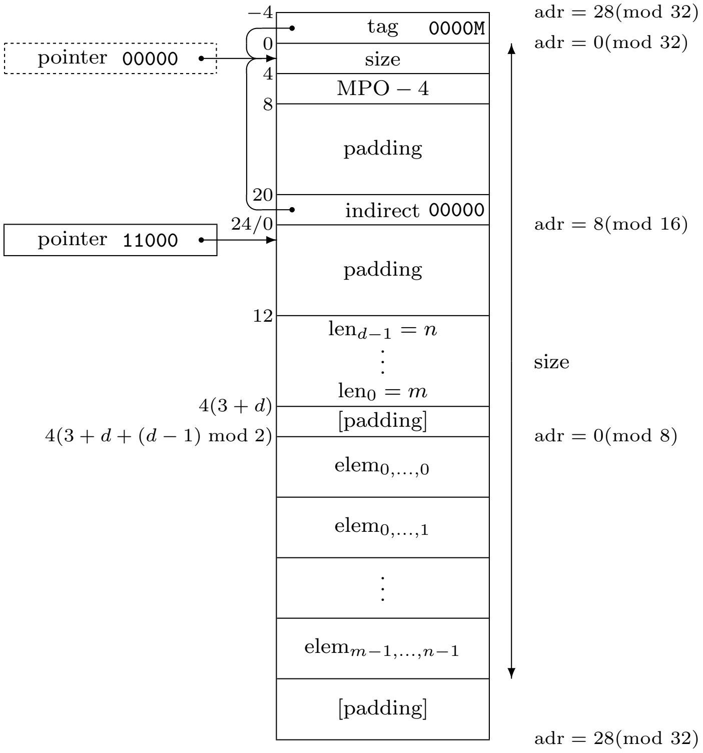


Figure B.4: A SysBlk is used for an array when the elements do not contain pointers.

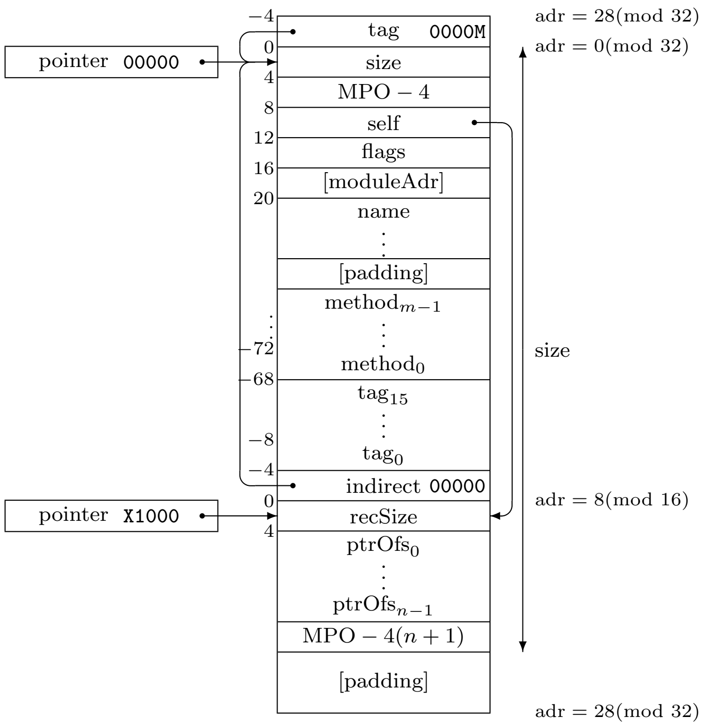


Figure B.5: A TypeDesc for a record type is similar to a SysBlk.

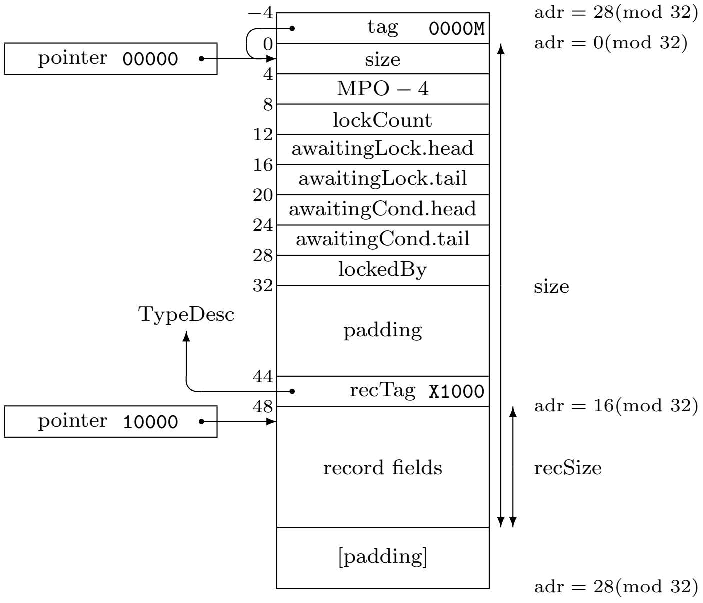


Figure B.6: A ProtRecBlk is used for a protected record.

Appendix C

Alternative Interrupt Handling Model

Section 4.3.3 describes a high-level interrupt handling model that was implemented in the kernel. Here an alternative model is presented. This model was not implemented, due to its complexity in comparison to the implemented model.

Await-Interrupt Model

It is appropriate to view a device driver as a *cyclic process* that awaits and reacts to interrupts [129]. The handler process communicates with its client processes using the normal interprocess communication facilities.

The *await-interrupt model* for interrupt handling uses active objects, augmented with the *await-interrupt* statement, which suspends the current process until a specified interrupt occurs.

Figure C.1 is an example of an await-interrupt model device driver for a device periodically producing unsolicited data. The device has two states: uninitialized and initialized, and generates an interrupt when it has been initialized. After initialization, the device can be programmed to generate an interrupt when unsolicited data becomes available. The driver is an active object, with a body for handling interrupts, and methods for its clients, who communicate with the interrupt handler via exclusive blocks and await statements (cf. 2.2.2).

```
TYPE Driver = OBJECT
  VAR "driver state" (* shared by client and driver processes *)

  PROCEDURE Read(...); (* client process calls this to read data *)
  BEGIN {EXCLUSIVE} (* mutually exclusive with driver process *)
    "return state information"
  END Read;

BEGIN {ACTIVE} (* this is the life-cycle of the driver process *)
  "initialize device"
  "await-interrupt" (* wait for device to respond *)
  LOOP
    "set up device to interrupt when data available"
    "await-interrupt" (* wait for device to generate interrupt *)
    BEGIN {EXCLUSIVE} (* mutually exclusive with client *)
      "get data from device and modify driver state"
    END
  END
END Driver;
```

Figure C.1: Active Oberon pseudo-code of a device driver using the await-interrupt model.

The main advantage of the await-interrupt model is that the device driver can be structured to directly reflect the state machine model of the device, without resorting to an event-driven programming style.

Non-Sharable Await-Interrupt Algorithm

Under the assumption that at most one process handles an interrupt (i.e., interrupts are not sharable between processes), the await-interrupt statement of the await-interrupt model can be implemented as a call to a runtime procedure that does the following:

1. The running process (the interrupt-handling process) is attached to the specified interrupt. This means that an entry is made in the interrupt vector table to associate the handler process with the interrupt.

2. The interrupt is unmasked by programming the interrupt controller.

3. The running process is suspended.

4. The scheduler is invoked to select another process to run, and a context switch is made to continue it from where it was last suspended.

When the specified interrupt occurs, the runtime system does the following:

1. The interrupt is masked. This is required to give the interrupt handler process time to remove the source of the interrupt signal. Otherwise, the interrupt would be signalled continually and will lock up the processor.

2. The suspended process is detached from the interrupt and enabled.

3. The scheduler is invoked, and if the newly enabled process has a sufficiently high priority, a context switch is made to it immediately. Otherwise, the interrupted process continues to run until it is suspended, or its time slice expires.

Even though it requires the masking of interrupts, this algorithm can also be applied to handle nonmaskable interrupts. A nonmaskable interrupt is similar to a normal interrupt, except that it has a very high priority, and can interrupt the processor even when it has disabled other interrupts. Despite the name, a nonmaskable interrupt can usually be masked by the interrupt controller (but not directly by the processor).

Sharable Await-interrupt Algorithm

Here the algorithm is generalized for the case where several processes can wait on the same interrupt (i.e., the interrupt is shared between different devices and their driver objects). We consider disjunctive sets of processes: each interrupt number has a *waiting set* and a *processing set*, and there is a global *satisfied set*.

- Initially all processes are in the *satisfied set*, meaning that they are not awaiting any interrupts.
- When a process in the satisfied set executes the AwaitInterrupt(n) call, it moves to the *waiting set* for interrupt n and is suspended until interrupt n next occurs.
- When interrupt n occurs, all processes in its waiting set are enabled and moved to its *processing set*. Then the scheduler is invoked, similar to the case described in the non-sharing algorithm. The handling processes can be scheduled to run in parallel on one or more processors.
- When a process in the processing set of interrupt n executes Await-Interrupt(m), it is moved to waiting set m, and suspended until interrupt m next occurs. In the common case, m is equal to n.
- When a process in the processing set of interrupt n terminates, it is moved to the satisfied set.

The reason for distinguishing the sets of processes is that the rule for masking interrupts can be stated as: Interrupt n is unmasked if, and only if, its waiting set is non-empty and its processing set is empty. In other words, an interrupt is unmasked if at least one process is waiting for it, unless some processes are still handling a previous occurrence. If a process enters the waiting set while the processing set is non-empty, it has no effect on the masking of the interrupt.

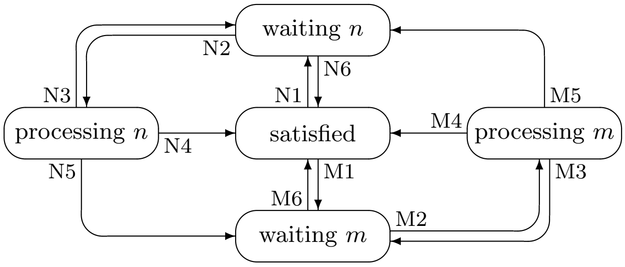


Figure C.2: State transitions in the active interrupt handling model.

The following data structures are used to implement the await-interrupt model efficiently:

- For every interrupt number there is a list of processes, and two integer counters each for the cardinality of the waiting set and the processing set, respectively. The list is initially empty, and the counters are zero. The list is used to store the processes in the waiting set and processing set.
- Every process has an integer storing its currently *bound interrupt*, which is initially -1, signifying that no interrupt is bound. Interrupt numbers are non-negative.

Figure C.2 shows all possible state transitions for interrupt handling. In reality, there are waiting and processing states for every possible interrupt number, but it suffices to describe two of each (n and m). The possible state transitions for interrupt n are:

N1 AwaitInterrupt(n) is executed by a process in the satisfied set.

N2 Interrupt n occurs.

N3 AwaitInterrupt(n) is executed by a process in processing set n.

N4 The process terminates itself.

N5 AwaitInterrupt(m) is executed by a process in processing set n.

	Waiting Counter	Processing Counter	Bound Interrupt	Scheduler and Process List
N1	$w'_n = w_n + 1$	ϕ	$b'_r = n$	suspend(r) $l'_n = l_n + r$
N2	$w'_n = 0$	$p'_n = w_n$	ϕ	$\forall q \in l_n$: enter(q)
N3	$w'_n = w_n + 1$	$p'_n = p_n - 1$	ϕ	suspend(r)
N4	ϕ	$p'_n = p_n - 1$	$b'_r = -1$	suspend(r) $l'_n = l_n - r$
N5	$w'_m = w_m + 1$	$p'_n = p_n - 1$	$b'_r = m$	suspend(r) $l'_n = l_n - r$ $l'_m = l_m + r$
N6	$w'_n = w_n - 1$	ϕ	$b'_r = -1$	$l'_n = l_n - q$

Figure C.3: State transition actions in the active interrupt handling model.

N6 The waiting process is terminated from outside by another process.

The state transitions for other interrupt numbers, M1–M6, etc., are symmetric to N1–N6.

Figure C.3 shows the actions corresponding to the state transitions for interrupt handling. w_n is the value of interrupt n's waiting counter before the transition, and w'_n is the value after the transition. Similarly, p_n is the value of the processing counter and l_n is the process list of interrupt n. r is the running process, and b_r is its bound interrupt number. The suspend(r) operation suspends the running process and the enter(q) operation enables suspended process q by entering it in the scheduler queue. ϕ indicates no operation.

The interrupt masking rule can now be paraphrased as: $M'_n = (w'_n = 0) \vee (p'_n \neq 0)$. The values of the counters before and after a transition are used in the implementation to decide whether to mask or unmask an interrupt.

The tabular notation precisely documents the implementation that was sketched at the start of this section. For example, transition N1 increments the waiting counter, binds interrupt n to the running process, suspends it and adds it to the list of interrupt n. This corresponds with a process being moved to the waiting set.

The AwaitInterrupt(n) (for any interrupt n) system call implements transitions N1, N3 and N5, so its first task is to determine which transition is relevant. N1 can be recognized by the running process's bound interrupt being -1. It is easy to see from figures C.2 and C.3 that this characterizes the satisfied state. N3 can be recognized by the bound interrupt being equal to the parameter n, and N5 is the case where the bound interrupt is not equal to the parameter n.

Transitions N2 and N4 are implemented straightforwardly by the interrupt handler procedure for interrupt n, and the process termination system call, respectively.

Transition N6 is required when cleaning up a device driver object. A special TerminateInterrupt system call can be provided for this purpose.

List of Figures

List of Tables

Bibliography

[1] 3Com Inc., 5400 Bayfront Plaza, Santa Clara, CA 95052-8145. *Etherlink III Parallel Tasking ISA, EISA, Micro Channel and PCMCIA Adapter Drivers Technical Reference*, 1994.

[2] 3Com Inc., 5400 Bayfront Plaza, Santa Clara, CA 95052-8145. *3C90x Network Interface Cards Technical Reference*, 1997.

[3] 3Dlabs Inc. *Permedia 2 Programmer's Reference Manual*, 1997.

[4] Adaptec Inc., 691 South Milpitas Blvd., Milpitas, California 95035. *SCSI ASIC Solutions: AIC-7890, AIC-7891, AIC-3860 Design-in Handbook*, 1997.

[5] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[6] G. A. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9), September 1990.

[7] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[8] ANSI. *T13-1153D Working Draft: Information Technology — AT Attachment with Packet Interface Extension (ATA/ATAPI-4)*, 1997.

[9] A. Baker and J. Lozano. *Gerätetreiber unter Windows 2000*. Markt+Technik Verlag, 2001. Translation of *The Windows 2000 Device Driver Book*, Prentice Hall PTR, 2001.

[10] G. Banfi. Just Another Way to Run Java Programs. Semester project, Department of Computer Science, ETH Zurich, 1997.

[11] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol — HTTP/1.0. RFC 1945, Network Working Group, 1996.

[12] B. N. Bershad, R. P. Draves, and A. Forin. Using Microbenchmarks to Evaluate System Performance. In *Proceedings of the Third Workshop on Workstation Operating Systems (WWOS-3)*, April 1992.

[13] H. Boehm. A Garbage Collector for C and C++. http://www.-hpl.hp.com/personal/Hans_Boehm/gc/.

[14] H. Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, 1993. *ACM SIGPLAN Notices*, 28(6), June 1993.

[15] H. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection. *ACM SIGPLAN Notices*, 26(6), 1991.

[16] J. Boykin and A. Langerman. The Parallelization of Mach/-4.3BSD: Design Philosophy and Performance Analysis. In *Distributed and Multiprocessor Systems Workshop.* USENIX Association, 1989.

[17] J. M. Bradshaw (ed.). *Software Agents.* The MIT Press, 1997.

[18] P. Brinch Hansen. Structured Multiprogramming. *Communications of the ACM*, 15(7), July 1972. Reprinted in [22].

[19] P. Brinch Hansen. *Operating System Principles.* Prentice-Hall, 1973.

[20] P. Brinch Hansen. Experience with Modular Concurrent Programming. *IEEE Transactions on Software Engineering*, 3(2), March 1977. Reprinted in [22].

[21] P. Brinch Hansen. Efficient Parallel Recursion. *ACM SIGPLAN Notices*, 30(12):9–16, December 1995. Reprinted in [22].

[22] P. Brinch Hansen. *The Search for Simplicity: Essays in Parallel Programming.* IEEE Computer Society Press, 1996.

[23] P. Brinch Hansen. *Classic Operating Systems.* Springer Verlag, 2001.

[24] T. Burri. Sound System für Oberon System 3 (Native Oberon). Semester project, Department of Computer Science, ETH Zurich, 1996.

[25] T. Burri. Development of a concurrent dynamic memory manager. Diploma thesis, Department of Computer Science, ETH Zurich, 1997.

[26] Computer Systems Institute, ETH Zurich. *Active Oberon for .NET.* http://www.oberon.ethz.ch/oberon.net/.

[27] R. Crelier. TDescs on the Heap and Subobjects (DECOberon heap blocks). Unpublished sketch, Computer Systems Institute, ETH Zurich.

[28] O. J. Dahl. Monitors Revisited. In A.W. Roscoe, editor, *A Classical Mind — Essays in Honour of C.A.R. Hoare.* Prentice-Hall, 1994.

[29] E. W. Dijkstra. The Structure of the 'THE' Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968. Reprinted in [23].

[30] A. R. Disteli. *Integration aktiver Objekte in Oberon am Beispiel eines Serversystems.* PhD thesis, Computer Systems Institute, ETH Zurich, 1997.

[31] A. R. Disteli and P. Reali. Combining Oberon with Active Objects. In [72], 1997.

[32] R. Droms. Dynamic Host Configuration Protocol. RFC 2131, Network Working Group, 1997.

[33] M. Dubois and C. Scheurich. Synchronization, Coherence and Event Ordering in Multiprocessors. *IEEE Computer*, February 1988.

[34] B. Egger. Development of an Aos Operating System for the DNARD Network Computer. Diploma thesis, Department of Computer Science, ETH Zurich, August 2001.

[35] B. Egger. Simple Installation and Windows Interoperability for ETH Oberon. Semester project, Department of Computer Science, ETH Zurich, 2001.

[36] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2068, Network Working Group, 1997.

[37] M. Franz. *Code-Generation On-the-Fly: A Key for Portable Software.* PhD thesis, Computer Systems Institute, ETH Zurich, 1994.

[38] M. Franz. Oberon — The Overlooked Jewel. In L. Böszörményi, J. Gutknecht, and G. Pomberger, editors, *The School of Niklaus Wirth.* dpunkt.verlag, 2000.

[39] M. Frei. Konzipierung eines Display Systems für Active Oberon. Semester project, Department of Computer Science, ETH Zurich, 1999.

[40] T. Frey. A Modern Graphical User Interface for the Aos System. Unpublished report, Computer Systems Institute, ETH Zurich, January 2002.

[41] T. Frey, H. Högger, O. Joos, and P. Kramer. Development of an Oberon System for the hyperstone E1 CPU. Semester project, Department of Computer Science, ETH Zurich, 1998.

[42] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Third Symposium on Operating System Design and Implementation (OSDI '99).* USENIX Association, 1999.

[43] R. Griesemer. On the Linearization of Graphs and Writing Symbol Files. Yellow Report 156, Computer Systems Institute, ETH Zurich, March 1991. Published with [85].

[44] J. Gutknecht. Oberon, Gadgets and Some Archetypal Aspects of Persistent Objects. Technical Report 243, Computer Systems Institute, ETH Zurich, February 1996.

[45] J. Gutknecht. Do the Fish Really Need Remote Control? A Proposal for Self-Active Objects in Oberon. In [72], 1997.

[46] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997. ACM Operating Systems Review 31(5), December 1997.

[47] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[48] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[49] F. Hrebabetzky. Closed-Loop Control with Oberon. *Software — Concepts and Tools*, 18:73–79, 1997.

[50] IECC. The Garbage Collection List. Mailing list. Archive at http://www.iecc.com/gclist/.

[51] Intel Corp. *Pentium Pro Family Developer's Manual Volume 3: Operating System Writer's Guide*, 1996.

[52] Intel Corp. *82371 FB (PIIX) and 82371SB (PIIX3) PCI ISA IDE Xcelerator Data Book*, 1997.

[53] Intel Corp. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2001. Order Number 245472, http://developer.intel.com/.

[54] Internet Engineering Task Force, Network Working Group. *Internet Official Protocol Standards*, 2001. RFC 3000.

[55] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, 1996.

[56] J.L. Keedy. On Structuring Operating Systems With Monitors. *ACM Operating Systems Review*, 13(1), January 1979.

[57] C. Kleiner. Four Dimensional Dreams. Diploma thesis, Department of Computer Science, ETH Zurich, March 1999.

[58] J. Kreienbühl. VNC Viewer for Oberon. Semester project, Department of Computer Science, ETH Zurich, 1999.

[59] F. Kuhn. Accellerated Display Driver and 3D Graphics Library for the Permedia 2 chip. Semester project, Department of Computer Science, ETH Zurich, 1999.

[60] S. Lalis and B. A. Sanders. Adding Concurrency to the Oberon System. In J. Gutknecht, editor, *Lecture Notes in Computer Science 782: Programming Languages and System Architectures*, pages 328–344. Springer Verlag, March 1994.

[61] B. W. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[62] A.M. Lister and P.J. Sayer. Hierarchical Monitors. *Software — Practice and Experience*, 7:613–623, 1977.

[63] J. Marais. *Design and Implementation of a Component Architecture for Oberon.* PhD thesis, Computer Systems Institute, ETH Zurich, 1996.

[64] J. Mauro and R. McDougall. *Solaris Internals.* Prentice-Hall, 2001.

[65] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, 1996.

[66] B. Meyer. Systematic Concurrent Object-Oriented Programming. *Communications of the ACM*, 36(9), September 1993.

[67] P. Mockapetris. Domain Names — Concepts and Facilities. RFC 1034, Information Sciences Institute, University of Southern California, 1987.

[68] P. Mockapetris. Domain Names — Implementation and Specification. RFC 1035, Information Sciences Institute, University of Southern California, 1987.

[69] D. Mosberger. Memory Consistency Models. Technical Report TR 93/11, Department of Computer Science, The University of Arizona, 1993.

[70] H. P. Mössenböck, J. Templ, and R. Griesemer. Object Oberon: An Object-Oriented Extension of Oberon. Yellow Report 109, Computer Systems Institute, ETH Zurich, June 1989.

[71] H. P. Mössenböck and N. Wirth. The Programming Language Oberon-2. *Structured Programming*, 12(4), 1991.

[72] H. P. Mössenböck (ed.). *Lecture Notes in Computer Science 1204: Proceedings of the Joint Modular Languages Conference, JMLC'97*. Springer Verlag, March 1997.

[73] D. Müller. Development of a 64-bit MIPS Implementation of the Oberon System. Diploma thesis, Department of Computer Science, ETH Zurich, 1997.

[74] P. J. Muller. Native Oberon Operating System. Web site. http://-www.oberon.ethz.ch/native/.

[75] P. J. Muller and P. J. A. de Villiers. Using Oberon to Design a Hierarchy of Extensible Device Drivers. In P. Schulthess, editor, *Proceedings of the Joint Modular Languages Conference, JMLC'94*, Ulm, Germany, September 1994. Universitätsverlag Ulm.

[76] Olivetti Research. *Virtual Network Computing*. http://www.uk.-research.att.com/vnc/.

[77] E. Oswald. *A Generic 2D Graphics API with Object Framework and Applications*. PhD thesis, Computer Systems Institute, ETH Zurich, 2000.

[78] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2), March 1979.

[79] D. L. Parnas. Why Software Jewels Are Rare. *IEEE Computer*, 29(2), February 1996.

[80] D. L. Parnas and D. P. Siewiorek. Use of the Concept of Transparency in the Design of Hierarchically Structured Systems. *Communications of the ACM*, 18(7), July 1975.

[81] D.L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.

[82] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.2*, 1998. http://www.pcisig.com/.

[83] J. K. Peacock. File System Multithreading in System V Release 4 MP. In *USENIX Summer 1992 Technical Conference*, pages 19–29. USENIX Association, June 1992.

[84] F. V. Peschel. *Vamos — Entwurf und Realisierung eines erweiterbaren Betriebssystems für Arbeitsplatzrechner*. PhD thesis, Computer Systems Institute, ETH Zurich, 1989.

[85] C. Pfister (ed.), B. Heeb, and J. Templ. Oberon Technical Notes. Yellow Report 156, Computer Systems Institute, ETH Zurich, March 1991. Published with [43].

[86] R. Pike. Systems Software Research is Irrelevant. Presentation at the Symposium on Operating Systems Principles (SOSP), February 2000. http://www.cs.bell-labs.com/who/rob/utah2000.pdf.

[87] C. Plattner. Universal Serial Bus Unterstützung für ETH Oberon. Semester project, Department of Computer Science, ETH Zurich, 2000.

[88] D. C. Plummer. An Ethernet Address Resolution Protocol. RFC 826, Network Working Group, 1982.

[89] J. Postel. Echo Protocol. RFC 862, Information Sciences Institute, University of Southern California, 1983.

[90] J. Postel (ed.). User Datagram Protocol. RFC 768, Information Sciences Institute, University of Southern California, 1980.

[91] J. Postel (ed.). Internet Control Message Protocol. RFC 792, Information Sciences Institute, University of Southern California, 1981.

[92] J. Postel (ed.). Internet Protocol. RFC 791, Information Sciences Institute, University of Southern California, 1981.

[93] J. Postel (ed.). Transmission Control Protocol. RFC 793, Information Sciences Institute, University of Southern California, 1981.

[94] P. Reali. Jaos. Web site. http://www.oberon.ethz.ch/jaos/.

[95] P. Reali. Structuring a Compiler with Active Objects. In *Joint Modular Languages Conference*. Springer Verlag, September 2000.

[96] P. Reali. *Language Interoperability.* PhD thesis, Computer Systems Institute, ETH Zurich, 2002.

[97] Realtek Semiconductor Corp. *Realtek RTL8139(A/B) Programming Guide*, 1999.

[98] M. Reiser and N. Wirth. *Programming in Oberon: Steps beyond Pascal and Modula.* Addison-Wesley, 1992.

[99] Digital Research. The DIGITAL Network Appliance Reference Design. Web site. http://www.research.compaq.com/SRC/iag/.

[100] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1), January/February 1998.

[101] T. Richardson and K. R. Wood. The RFB Protocol. Technical report, Olivetti Research Lab, Cambridge, 1998.

[102] R.Laich. Eine Java Virtual Machine für Aos. Diploma thesis, Department of Computer Science, ETH Zurich, 2001.

[103] M. Russinovich. Win2K Queued Spinlocks. http://www.compuware.com/products/driverstudio/resources/% -papers/spinlocks.-htm.

[104] S3 Inc., Santa Clara, CA 95052-8058. *Trio64V+ Integrated Graphics/Video Accelerator*, 1996.

[105] R. W. Scheifler, J. Gettys, D. Converse, and A. Mento. *X Window System: Core Libraries and Standards.* Butterworth-Heinemann, 1996.

[106] C. Schimmel. *Unix Systems for Modern Architectures.* Addison-Wesley, 1994.

[107] H. Schorr and W. Waite. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *Communications of the ACM*, 10(8), August 1967.

[108] M. L. Scott, T. J. LeBlanc, B. D. Marsh, T. G. Becker, C. Dubnicki, E. P. Markatos, and N. G. Smithline. Implementation Issues for the Psyche Multiprocessor Operating System. *Computing Systems*, 3(1), Winter 1990.

[109] D. Seal (ed.). *ARM Architecture Reference Manual.* Addison-Wesley, second edition, 2001.

[110] J. Sedlacek. Project C2: A Survey of an Industrial Embedded Application with PC Native Oberon. In *Joint Modular Languages Conference.* Springer Verlag, September 2000.

[111] A. Signer. Development of an Oberon System 3 for the DNARD Network Computer. Diploma thesis, Department of Computer Science, ETH Zurich, 1999.

[112] A. Silberschatz and P. B. Galvin. *Operating System Concepts.* Addison-Wesley, fifth edition, 1998.

[113] W. Simpson (ed.). The Point-to-Point Protocol (PPP). RFC 1661, Network Working Group, 1994.

[114] D. A. Solomon and M. E. Russinovich. *Inside Microsoft Windows 2000.* Microsoft Press, third edition, 2000.

[115] R. Strobl. Optimizing the Native Oberon NetSystem. Semester project, Department of Computer Science, ETH Zurich, 1999.

[116] P. Stüdi. Ein Client-Server Filesystem für Aos. Semester project, Department of Computer Science, ETH Zurich, 2001.

[117] J. Supcik. HP-Oberon – The Oberon Implementation for HP 9000 Series 700. Yellow Report 212, Computer Systems Institute, ETH Zurich, 1994.

[118] C. A. Szyperski. *Insight ETHOS: On Object-Orientation in Operating Systems.* PhD thesis, Computer Systems Institute, ETH Zurich, 1992.

[119] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[120] J. Templ. *Metaprogramming in Oberon.* PhD thesis, Computer Systems Institute, ETH Zurich, 1994.

[121] C. P. Thacker and L. C. Stewart. Firefly: A Multiprocessor Workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987.

[122] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. *The Journal of Supercomputing*, 9:105–136, 1995.

[123] USB Org. *Universal Serial Bus (USB) Specification Rev. 2.0*, 2000. http://www.usb.org/.

[124] F. van Gilluwe. *The Undocumented PC: A Programmer's Guide.* Addison-Wesley, second edition, 1997.

[125] Video Electronics Standards Association, 2150 North First Street, Suite 440, San Jose, CA 95131-2029. *Vesa BIOS Extension (VBE) Core Functions Version 2.0.*

[126] C. von Praun and T. Gross. Compiler-based Object Consistency. In *Workshop on Caching, Coherency and Consistency (WC3 '01)*, June 2001.

[127] S. Walthert. Entwicklung eines Style Layers und Renderers für die XML-basierte GUI-Shell des Aos Systems. Diploma thesis, Department of Computer Science, ETH Zurich, 2001.

[128] D. A. Wheeler. More Than a Gigabuck: Estimating GNU/Linux's Size. http://www.dwheeler.com/sloc/.

[129] N. Wirth. Modula: A Language for Modular Multiprogramming. *Software — Practice and Experience*, 7:3–35, 1977.

[130] N. Wirth. The Use of Modula. *Software — Practice and Experience*, 7:37–65, 1977.

[131] N. Wirth. *Programming in Modula-2.* Springer Verlag, third, corrected edition, 1985.

[132] N. Wirth. The Programming Language Oberon. *Software — Practice and Experience*, 18(7):671–690, July 1988.

[133] N. Wirth. A Plea for Lean Software. *IEEE Computer*, February 1995.

[134] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

[135] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated*, volume 2. Addison-Wesley, 1995.

[136] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6), June 1974.

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

[138] B. G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, December 1989.

Curriculum Vitae

Pieter Johannes Muller

11th July 1968 Born in Nababeep, South Africa. Son of Andries and Coerie Muller.

1975–1983 High School Nababeep, South Africa.

1984–1986 *Paul Roos Gimnasium*, Stellenbosch, South Africa.

1987–1989 B.Sc. Mathematical Sciences, Stellenbosch University, South Africa.

1990–1991 B.Sc. Hons. Computer Science, Stellenbosch University, South Africa.

1991–1994 M.Sc. (cum laude) Computer Science, Stellenbosch University, South Africa.

1991–1995 Research assistant, *Instituut vir Toegepaste Rekenaarwetenskap*, Stellenbosch University, South Africa.

1995–2001 Research and teaching assistant, Computer Systems Institute, ETH Zurich.

2002 Research and development engineer, MCT Lab GmbH, Zurich.