Sony Computer Science Labor
Technical Memo
SCSL-TM-94-005

Concurrent Object-Oriented Device Driver Programming in *Apertos* Operating System

Jun-ichiro Itoh * Yasuhiko Yokote †

August 22, 1994

Abstract

This paper proposes a new approach of implementing low-level OS components, especially device drivers. We introduce the notion of concurrent objects into device driver programming. A device driver for every hardware device is implemented as independent concurrent objects. A device driver object has a single thread of control, so that mutual exclusion operations such as `spl` and semaphores are not necessary. Mechanisms of synchronization, scheduling objects, and interrupt mask handling are clearly separated from actual device control programs, and these are implemented by the system for device driver objects. Both an interrupt and a service request to a device driver object will be delivered as a message, so programmers need not to distinguish these two types of invocation requests. Therefore, programmers can concentrate on writing the actual device control codes without writing auxiliary codes for synchronization, and the codes will be executed much safer. We show our implementation on *Apertos* object-oriented operating system, and its preliminary evaluation results. Evaluation results demonstrate our approach can be implemented cost-effective.

1 Introduction

In these days, demands on highly distributed and heterogoneous computing are increasing. This is driven by the trends of building Information Super Highway. Particularly, VOD (Video on Demand) services using continuous-media servers and settop-boxes interconnected by networks will be available in the near future. Settop-boxes and other embedded equipments such as VCR's, TV's, PDAs, and game machines have characteristics such as limitation of system resources, real-time constraints, and at the same time distribution and heterogeneity. Although existing embedded applications are implemented on top of some embedded operating systems, such operating systems can hardly support the high abstractions, like distribution. Thus, we claim that future embedded operating systems should provide:

- dynamic creation/destruction of low-level system components,
- update and bugfixes of the entire system without stopping the system, and
- programming and execution models optimal for various applications' demands such as distribution, persistence, location transparency, heterogeneity, and atomicity which are applicable to low-level system codes through high-level application codes.

Recently, mobile computers with PCMCIA interfaces and some network controllers support the hot-swap facility, so that the system should provide mechanisms to dynamilcally install/remove low-level system components from secondary storage or networks. This enables us to make a room for applications unless low-level system components are installed. This also helps us to their runtime update and bugfixes. In this paper, the discussion mainly goes to device driver programming because of its importance.

In many existing operating systems, device drivers are implemented inside their kernel, and cannot be replaced. Also, it is difficult to implement (and debug) device drivers, because:

*Keio University, Department of Computer Science, Tokoro Laboratory. 3-14-1, Hiyoshi, Kohoku-Ku, Yokohama, Kanagawa 223 JAPAN. e-mail: `itojun@mt.cs.keio.ac.jp`

†Sony Computer Science Laboratory. Takanawa Muse Buliding, 3-14-13, Higashi-Gotanda, Shinagawa, Tokyo 141 JAPAN. e-mail: `ykt@csl.sony.co.jp`

- codes for device control, scheduling, and mutual exclusion are mixed up;
- the unit of mutual exclusion is unclear; and
- operations for scheduler and mutual exclusion are complicated, so that irregular usage of these operations causes the system hang-up.

We believe that the dynamic change of device drivers can only be achieved by implementing them completely independent. If they are mutually dependent, the change of a driver will affect the others and may cause troubles, i.e., system hang-up. Therefore, we need a more elaborate programming model to make device drivers independent, and make their implementation easy.

In this paper, we propose the following two design strategies in device driver programming:

- a concurrent object: a single-threaded modeule is a unit of execution.
- separation of concerns: codes for synchronization, scheduling, and interrupt mask control are separated from device control codes.

Using these design strategies, each device driver is completely independent, and programmers can concentrate their attention on device control programming. The mechanisms necessary to support these strategies are implemented in the *Apertos* operating system, and preliminary evaluation is shown in the later section of this paper.

2 Device Drivers as Concurrent Objects

We implement a device driver as a concurrent object[Yonezawa and Tokoro 87]. In this paper, a concurrent object is defined in such a way that it is an object having memory region for data, methods (or codes), and a single virtual processor (or a thread). A concurrent object has the following features:

- A method is invoked as an effect of message passing.
- An object is a unit of atomic execution, i.e., at most one method is executed at one time.

The first item helps us to treat an interrupt as message passing to a specific device driver object. Programmers need not to distinguish between them. A message sent by interrupt can be viewed as an asynchronous message with highest priority. The second item helps us to automatically create a mutual exclusive region, and make the region explicit to programmers, i.e., a concurrent object.

Introducing concurrent objects creates a problem of ordering two methods in the same object. That is, some device driver methods have to be invoked as an effect of the previous method execution. For example, reading data from a disk can be divided into two methods: one is requesting read operation and the other is receiving an acknowledgement of a finish operation. The latter method should be executed after the first method execution. In UNIX, functions `sleep` and `wakeup` are used for this purpose.

Since a pair of `sleep`/`wakeup` is error prone, we have decided not to allow such hard-coding of these operations, and we introduce *continuations* as an abstraction of scheduling operations. Here, a continuation is an object, and it is managed by the system[1]. A continuation forwards an asynchronous message to the object that the rest of its computation is represented by the continuation in order to invoke a specified method of the object. A continuation is created and destructed using an API provided by the system. Here is an example. Figure 1(a) shows a typical code fragment for functions `xxintr` and `xxread` in UNIX. These can be replaced with continuations as shown in Figure 1(b). `NewCont()` on line 12 creates a continuation which means `ReadCont()` have to be executed after completion of the `Read()` method. After creating a new continuation, method `Read()` is terminated in this example (line 12). Then, `Send()` on line 5 sends an asynchronous message to that continuation, and `ReadCont()` is invoked, as an effect of an interrupt. Operations on a continuation are executed safely than a pair of `sleep`/`wakeup`, because there is no way to put an erroneous code in device drivers.

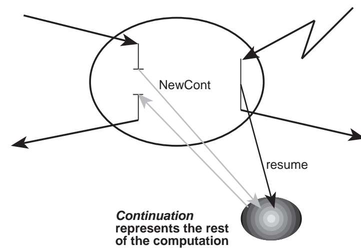
3 Separation of Concerns and the *Apertos* Operating System

Separation of concerns is a programming strategy which thinks about two concerns separately: ideal abstraction and mapping to efficient implementation[Yokote *et al.* 94]. Programmers want to concentrate programming of a given problem. Here, assume we have a black-box on top of which applications are constructed using the API

[1] We say a continuation is a *metaobject* in the sense that it is an object representing the rest of object's computation at its meta-level.

```
 1 xxintr()
 2 {
 3         read from device;
 4         wakeup(x);
 5 }
 6
 7 xxread()
 8 {
 9         if (notready)
10                 sleep(x);
11         return the result;
12 }
```

(a) UNIX style




```
 1 XX::Interrupt()
 2 {
 3         read from device;
 4         /*wakeup the continuation*/
 5         Send(cont, null);
 6 }
 7
 8 XX::Read()
 9 {
10         if (notready) {
11                 /*create a continuation*/
12                 cont = NewCont(ReadCont);
13                 Exit();
14         }
15         return the result;
16 }
17
18 XX:ReadCont()
19 {
20         /*reactivated*/
21         return the result;
22 }
```

(b) our style using continuation

Figure 1: Coding styles of the synchronization operation.

provided by that black-box. This is sometimes happy for application programmers because applications are free from the underlying implementation, so that they are portable. In traditional operating systems, for example, a process is an abstraction of the underlying implementation of scheduling, allocation to a physical processor, etc. In some cases, however, the black-box creates troubles of violating applications' demands such as performance and memory requirements. This implies programmers have to write codes optimal for their applications, but the codes are sometimes duplication of the ones implemented in the black-box. Also, programmers may inspect the internals of the black-box and modify them in order to meet applications' demands. This makes programs unsafe and less portable.

Apertos operating system[Yokote 92] provides systematic support of separation of concerns using the object/metaobject separation technology. In *Apertos*, every computational resource is defined as concurrent object. User applications, schedulers, network protocol handlers, device drivers, and other components organizing the system are implemented as independent concurrent objects. Therefore, concurrent objects are given an ideal abstraction and interface (API) to write applications, which we call base-level interface. Also, the internals of the black-box is exposed through another interface, which we call meta-level interface, and they are implemented by concurrent objects (or metaobjects). In this sense, we call the black-box a *metaspace*. Since a metaspace has the knowledge of characteristics and statistics of base-level objects' behavior, run-time optimization can be done by the metaspace. Also, a metaspace rejecting unsafe requests from base-level objects makes the system safe.

In case of device driver programming, programmers want to concentrate writing codes for device control. The implementation details of message passing, mutual exclusion, and scheduling should be hidden from codes written by programmers. In *Apertos*, device drivers are written using the base-level interface and implemented as concurrent objects. Metaobjects and a metaspace are introduced to provide device driver objects with optimal execution environment implementing the operations such as message passing, mutual exclusion, and scheduling. In traditional operating systems where these operations are hard-coded into device drivers, only a small mistakes causes disastrous conditions, i.e., system hang-up. Our approach can avoid such disastrous conditions, because programs using the base-level interface have no chance to write the operations.

In *Apertos*, a metaspace is realized by an metaobject called a *reflector*. A reflector accepts the requests from the objects residing on the metaspace implemented by the reflector, and it processes requests in cooperation with other metaobjects. Since a reflector is also an object, there are a metaspace for a reflector. In this sense, giving

an object causes metaspaces to construct their metahierarchy. An object can switch its metaspace, by invoking base-level interface `Migrate`. Since every metaspace supports it, every object can change the behavior and its representation, by moving from a metaspace to another.

Apertos implements four primitive operations. These are a primitive invoking a service provided by a metaspace (M), a primitive returning the result from the metaspace to an object (R), a primitive registering interrupt message delivery (CBind), and a primitive removing the registration made by CBind (CUnbind). *Apertos* also implements a very small micro-kernel called ***MetaCore***, which is located per CPU basis. No memory allocation or virtual memory management is done by ***MetaCore***. These are done by the low-level objects outside of the ***MetaCore***, and they are replaceable. ***MetaCore*** is the only non-replaceable entity in the *Apertos* system.

4 The Design of the Metaspace for Device Drivers

We introduce the metaspace for device driver objects, which is called m^{Drive}. The scheduling operations and mechanism for guaranteeing objects' atomicity are clearly separated from device driver objects. m^{Drive} provides the following interface to the device driver objects:

Inter-object communication: Synchronous message communication (RPC style, `Call`), asynchronous message communication (one-way, `Send`), and replying a result to the message sender (`Reply`) are provided.

Scheduler operations via continuations: Creating continuation (`NewContinuation`) and deleting continuation (`DeleteContinuation`) are available as described in Section 2. Also, a method is terminated without returning a result to the message sender (`Exit`).

Memory management: Memory region is allocated (`Grow`) and freed (`Shrink`).

Switching metaspace: Metaspace is switched using `Migrate` as described in Section 3.

4.1 Realtime constraints of m^{Drive}

Device drivers are usually time critical/dependent. If method execution of a device driver cannot meet its realtime constraints, disastrous conditions may occur. At the same time, it is difficult for guaranteeing realtime connstraints to be statically analyzed, because an interrupt is completely a sporadic event. To guarantee to meet the realtime constraints, the following restrictions should be applied to the design of m^{Drive}:

- All the above operations should be predictable.
  Operations such as memory allocation and continuation creation should complete its execution in the exact time period. Also, memory pages should be pin-downed to the physical memory space to avoid unnecessery access delay caused by paging.

- Device driver objects on m^{Drive} should not issue synchronous message send (i.e., `Call`) to the target object which is not supported by m^{Drive}.
  We do not allow synchronous calls to the outside of m^{Drive}. Otherwise, completion of device driver execution cannot be bounded, since a device driver object can potentially make a synchronous call to the target object that is not aware of its execution time period. Thus, we have designed m^{Drive} to disallow such calls.

4.2 Execution overhead

Our approach might have the severe execution overhead, because invoking independent device driver objects may need full context-switch. In the current implementation, however, the overhead is very low, and it can be reduced by introducing run-time optimization including context-switch avoidance when it is needed and safe to do so.

In *Apertos*, every aspect of an object such as protection, activity, and persistence can be defined by designing a new metaspace. For example, a user application can be given its own address space, arbitrary number of execution thread scheduled with a preemptive scheduler, and persistence transparency. Likewise, a low-level object of an operating system can share its address space with other objects, provide the single execution thread scheduled by a non-preemptive scheduler, and has no persistence.

We have designed m^{Drive} so that the processor context-switch is the only operation needed to switch the execution from a device driver object to another. Switching an address space is not necessary. Every processor context-switch is done without trap instruction[Yokote 93] and its cost is low. Preliminary evaluation of our implementation is presented in the next section.

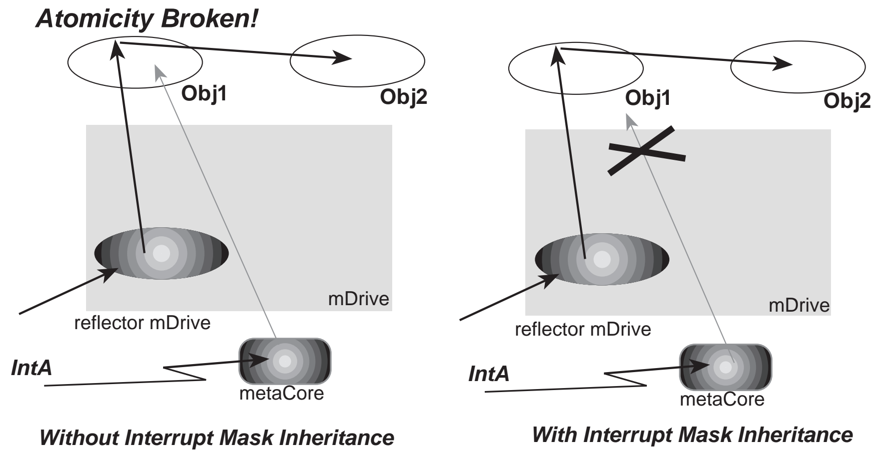


Figure 2: Interrupt mask inheritance between two objects.

4.3 Interrupt mask control

How interrupt masks be controlled to guarantee atomicity of a device driver object? In Figure 2, if device driver object ***Obj1*** receives interrupt ***IntA***, ***IntA*** should be masked while one of the methods of ***Obj1*** is executing. Also, ***IntA*** should be masked while a method of object ***Obj2*** is invoked by ***Obj1*** via synchronous call (`Call`). If ***IntA*** is not masked, two methods of ***Obj1*** will be executed simulaneously when ***IntA*** occurs, and ***Obj1*** will become inconsistent state. Therefore, an interrupt mask for execution of ***Obj1*** should make ***IntA*** masked. Also, ***Obj2*** should inherit the interrupt mask from ***Obj1*** while its method execution. Such interrupt mask control is managed by m^{Drive}, so the programmers can be ignorant of the interrupt mask control.

4.4 Dynamic creation and deletion of device drivers

Apertos provides transparent persistent object storage[Tenma *et al.* 92] and the transparent class system. These are implemented by metaspaces, and switching metaspace means the change of an object's representation and behavior. Thus, switching the metaspace for a device driver object with another metaspace supporting persistence makes the object freeze or activate into/from secondary storage. Otherwise, we can create a new device driver object by its class system, i.e., by switching a metaspace creating a new object with metaspace m^{Drive}.

5 Preliminary Evaluations

m^{Drive} has been implemented on i486-based PC-AT compatible machines. The basic cost of *Apertos* primitives is summarized in Table 1[2]. Table 2 shows the cost of the operations related to execution of device driver objects. The

Table 1: Execution cost of *Apertos* primitives. (in μsec)

primitive	*on i486*
M	21.1
M(w/o trap)	13.0
R	22.6
R(w/o trap)	8.8
CBind	4.1
CUnbind	3.5

Current version of the m^{Drive} does not utilize optimization techniques such as context-switch avoidance. Figure 3 shows the execution flow of the null interrupt handler.

[2]Evaluation results were obtained on Sony PCX-500V(Intel i486DX2, 66MHz, with 16MB physical memory)

Table 2: Costs of mDrive services, and interrupt operations. (in μsec)

operation	*on i486*
Call-Reply roundtrip on mDrive	207.8
Interrupt message delivery	25.0
Null interrupt handler execution	44.2
Send metacall overhead	108.6

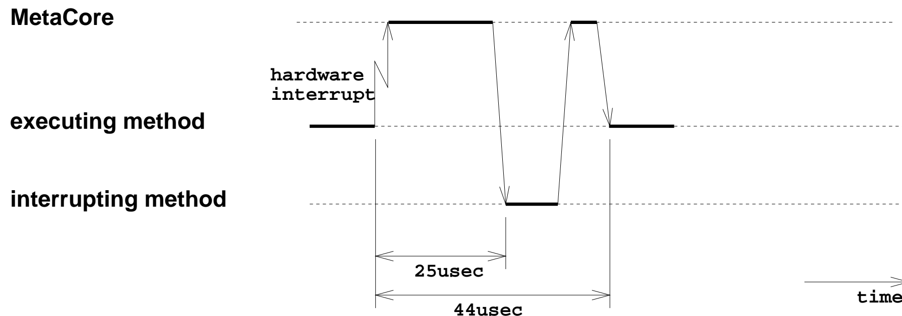


Figure 3: Execution flow of null interrupt handler.

The results of Table 2 enable us to estimate the cost incurred by an execution path of a typical interrupt handler when an interrupt occurs. A typical interrupt handler has the following operations:

1. transfering data to/from a hardware device, and
2. sending data to other objects as a message, or re-activates a continuation.

The cost involved in the first operation is difficult to estimate because it depends on a hardware device. The cost of the second operation takes 108.6μsec. (see Table 2). It includes the cost of invoking primitives M and R once each. By adding the cost of null interrupt handler execution, we can estimate the minimal execution time needed to execute a typical interrupt handler, which is 153μsec per interrupt. Figure 4 shows the execution flow of a typical interrupt handler.

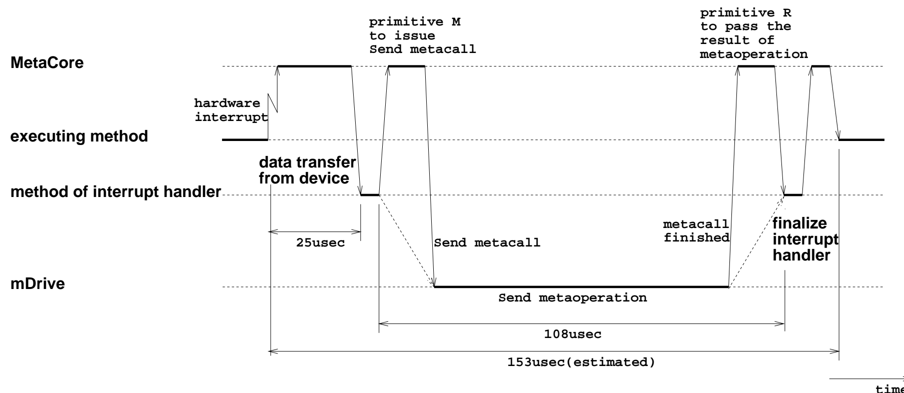


Figure 4: Execution flow of the typical interrupt handler.

We have designed mDrive not to hook interruptive messages which are sent from ***MetaCore*** to the device driver objects, to minimize the delay while invoking an interrupt handler. If we designed mDrive to hook interruptive messages (it might be easier to manage the state of the objects, if we hooked it), two more invocations of *Apertos*

primitives would be necessary while handling interruptive messages. In such an implementation, the null interrupt handler execution path will be 75μsec (calculated from the results in Table 1 and 2).

The micro-benchmark evaluation of primitive operations shows that the time consumed by loading/saving registers dominates their cost. In the current implementation, the biggest potion of the time taken to execute m^{Drive} operations is consumed by a hierarchical naming server[Fujinami and Yokote 92]. These costs can be minimized, by modifying register loading/saving methods and implementing a new naming server with realtime guarantee of its execution.

6 Related Work

Sony's NEWS-OS 4.0[Sony Corporation92] supports dynamic loading of device drivers using command `devattach`. NEWS-OS 6.0[Sony Corporation94] (SVR4.2 variants) also supports the same functionality. LKM (Loadable Kernel Module) implemented on SunOS[Sun Microsystems91] and NetBSD[NetBSD project94] enables kernel modules to be dynamically loaded using command `modload`. In these systems, a module loaded into the kernel is actually relocated and linked into the kernel memory space. Entry points to the module will be attached by modifying kernel memory space directly. Programmers have to invoke function calls for scheduler and mutual exclusion such as `spl`, `sleep`, and `wakeup`. Thus, programming style/discipline of the modules have to be highly documented, otherwise such mechanisms are error prone. Also, each device driver is mutually dependent through interrupt mask control and scheduler operations. If a newly loaded kernel module conflicts with others, it causes a disastrous situation. To make matters worse, the mechanism for loading kernel module is too dangerous.

Chorus[Rozier *et al.* 88] and Mach[Tevanian and Rashid 87] introduce user-level device drivers. In case of Chorus, [Armand 91] describes their device driver implementation as separate actors. Though a device driver actor communicates with other components using ports and messages, UNIX-style mutual exclusion manipulation is still needed to implement a device driver actor. Also, they did not mentioned problems on the time-bound guarantees and memory management policies for a device driver actor. Mach[Forin *et al.* 91] divides a device driver into two parts: hardware dependent and independent. It implements a device driver as an independent task. Mach device driver implementation needs special kernel codes for dispatching a thread which handles the interrupt. Also, a device driver task can potentially invoke any Mach kernel calls, therefore there is no fail-safeness. Both papers persist in implementing "user-level" device drivers. We claim that the most important point is how their execution can be guaranteed to be safe, with consideration of realtimeness, virtual memory management latency, etc.

Micro-computer operating systems used for embedded systems, such as OS-9 and MS-DOS, also have a mechanism to dynamically load a device driver. Though they do not address the above issues, their execution mechanisms are in ad-hoc basis, and they do not provide high abstractions such as distribution transparence to programmers.

7 Conclusion

In this paper, we have presented our new approach in device driver programming. We have clearly separated actual device control codes from other management codes such as scheduler operations, mutual exclusion, and communication between device drivers. We have implemented a device driver as single-threaded concurrent objects, and successfully reduced the frustrations of device driver writers. Preliminary evaluation has shown that our approach can be implemented sufficiently cost-effective.

We are continuing our research and will demonstrate how much competitive our approach than others in the final paper. Further, we have to address the following issues:

- More deep consideration to realtimeness, which includes message delivery time guarantee, elaborate realtime scheduler suitable for the *Apertos* meta-architecture and realtime naming manager.
- Reduction of execution cost which introduces new optimization techniques.
- Comparison of the execution cost versus other implementation of the device drivers including BSD UNIX, Mach, Chorus.
- Design of object protocols for hot-swap awareness. To implement hot-swap aware device drivers, we also have to implement the interrupt handler for interrupts from a PCMCIA controller, like PCMCIA card services and socket services implemented on MS-DOS[JEIDA and PCMCIA 93].

Acknowledgments

The authors would like to thank Takao Tenma and Nobuhisa Fujinami at SonyCSL, Koichi Moriyama at Sony, and Ken-ichi Murata at Keio University, for their helpful comments and suggestions.

References

[Armand 91] François Armand. Give a Process to your Drivers! In *Proc. of the EurOpen Autumn 1991 Conference*, Sep. 1991.

[Forin *et al.* 91] Alessandro Forin, David Golub, and Brian Bershad. An I/O System for Mach. In *Proceedings of the Usenix Mach Symposium*. USENIX, Nov. 1991.

[Fujinami and Yokote 92] Nobuhisa Fujinami and Yasuhiko Yokote. Naming and Addressing of Objects without Unique Identifiers. In *12th International Conferrence on Distributed Computing Systems*, June 1992. also appeared in Sony Computer Science Laboratory Inc., Technical Report SCSL-TR-92-004.

[JEIDA and PCMCIA 93] JEIDA and PCMCIA. *PC card guidelines*. JEIDA, March 1993. (in Japanese).

[Kiczales and Lamping 93] Gregor Kiczales and John Lamping. Operating Systems: Why Object-Oriented? In *Proceedings Third International Workshop on Object Orientation in Operating Systems(IWOOOS'93)*, pp. 25–30, Dec. 1993.

[Kiczales *et al.* 92] Gregor Kiczales, Marvin Theimer, and Brent Welch. A New Model of Abstraciton for Operating System Design. In *Proceedings Second International Workshop on Object Orientation in Operating Systems(IWOOOS'92)*, pp. 346–349, 1992.

[NetBSD project94] NetBSD project. *NetBSD-current source codes*, 1994.

[Rozier *et al.* 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. CHORUS Distribted Operating Systems. *Computing Systems*, Volume 1, No. 4, pp. 305–370, Fall 1988.

[Sony Corporation92] Sony Corporation. *NewsOS 4.0 Reference Manual*, 1992.

[Sony Corporation94] Sony Corporation. *NewsOS 6.0 System V Release 4.2 Device Driver Reference*, 1994.

[Sun Microsystems91] Sun Microsystems. *SunOS 4.1.3 Reference Manual*, 1991.

[Tenma *et al.* 92] Takao Tenma, Yasuhiko Yokote, and Mario Tokoro. Implementing Persistent Objects in the Apertos Operating System. In *Proceedings Second International Workshop on Object Orientation in Operating Systems(IWOOOS'92)*, pp. 66–79. IEEE, 1992.

[Tevanian and Rashid 87] Avadis Tevanian, Jr. and Richard F. Rashid. MACH: A Basis for Future UNIX Development. Technical Report CMU-CS-87-139, Computer Science Department, Carngegie Mellon University, June 1987.

[Yokote 92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *OOPSLA '92 Proceedings*, pp. 414–434. ACM, Oct. 1992. also appeared in Sony Computer Science Laboratory Inc., Technical Report SCSL-TR-92-014.

[Yokote 93] Yasuhiko Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS)*. JSSST, November 1993. also appeared in Sony Computer Science Laboratory Inc., Technical Report SCSL-TR-93-014.

[Yokote *et al.* 94] Yasuhiko Yokote, Gregor Kiczales, and John Lamping. Separation of Concerns and Operating Systems for Highly Heterogeneous Distributed Computing. In *6th ACM SIGOPS European Workshop*. ACM SIGOPS, July 1994. also appeared in Sony Computer Science Laboratory Inc., Technical Report SCSL-TR-94-022.

[Yonezawa and Tokoro 87] Akinori Yonezawa and Mario Tokoro, editors. *Object Oriented Concurrent Programming*. The MIT Press, 1987.