THE DESIGN AND APPLICATION OF AN EXTENSIBLE OPERATING SYSTEM

Leendert van Doorn

VRIJE UNIVERSITEIT

THE DESIGN AND APPLICATION OF AN EXTENSIBLE OPERATING SYSTEM

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen /
Wiskunde en Informatica
op donderdag 8 maart 2001 om 10.45 uur
in het hoofdgebouw van de universiteit,
De Boelelaan 1105

door

LEENDERT PETER VAN DOORN

geboren te Drachten

Promotor: prof.dr. A.S. Tanenbaum

To Judith and Sofie

Publisher: Labyrint Publication
P.O. Box 662
2900 AR Capelle a/d IJssel - Holland
fax +31 (0) 10 2847382

ISBN 90-72591-88-7

Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 60.

Parts of Chapter 2 have been published in the *Proceedings of the First ASCI Workshop* and in the *Proceedings of the International Workshop on Object Orientation in Operating Systems*.

Parts of Chapter 3 have been published in the *Proceedings of the Fifth Hot Topics in Operating Systems (HotOS) Workshop*.

Parts of Chapter 5 have been published in the *Proceedings of the Sixth SIGOPS European Workshop*, the *Proceedings of the Third ASCI Conference*, the *Proceedings of the Ninth Usenix Security Symposium*, and filed as an IBM patent disclosure.

Contents

Acknowledgments

Although my advisor, Andy Tanenbaum, thinks otherwise, I view a Ph.D. as a period of learning as much as you can in as many *different* subjects that are interesting. In this respect I took full advantage of my Ph.D.: I did work ranging from programming my own EEPROMs, to secure network objects, digital video on demand, and a full blown new operating system with its own TCP/IP stack, an experimental Orca runtime system, and a secure Java virtual machine. I even considered building my own network hardware but was eventually persuaded to postpone this. This advice no doubt sped up the completion of this thesis considerably.

There are a large number of people who assisted and influenced me during my Ph.D. and for which I have great admiration. First of all there is my advisor, Andy Tanenbaum, who was always quick to jump on my half-baked ideas and forced me to give better explanations, and of course Sharon Perl who picked me as an intern at Digital Systems Research Center. I really felt at home at SRC, in fact so much that I came back the next summer to work with Ted Wobber on secure network objects. Two other SRC employees who deserve an honorable mention are Mike Burrows for our technical discussions and for pointing out to me that puzzles are actually fun, and Martin Abadi for his conciseness and precision; Virtues I often lack.

Of course, I should not forget Rob Pike, who thought I was spending too much time at DEC SRC and that I should also visit Bell Labs. I was more than eager to take him up on that. During that summer I had ample opportunity to play with Inferno, Plan 9 and digital video. I'm forever in debt to Ken and Bonnie Thompson for their hospitality, which still extends to today. Most of my time at Bell Labs I spent with Dave Presotto and Phil Winterbottom. Phil reminded me in more than one way of Mike Burrows, albeit a much more critical version.

As so many Ph.D. students, I took longer than the officially approved four years to finish my thesis. Rather than staying at the university, I was persuaded to join IBM T.J. Watson Research Center as a visiting scientist and finish my thesis there. After four months it was clear I liked the place; industrial research laboratories are surprisingly similar; and I joined IBM as a research staff member. Here I worked on my secure Java Virtual Machine, but also got distracted enough that the writing of my thesis got delayed considerably, despite the almost daily reminders by my group members and not infrequent ones from my advisor. My group members included Charles Palmer, Dave Safford, Wietse Venema, Paul Karger, Reiner Sailer and Peter Gutmann. I managed to tackle this daily nuisance by stating that inquiries about my thesis progress were in fact voluntary solicitations to proofread my thesis. Some, who did not get this in time, ended up proof reading my drafts. Then there were others who practically forced me to give them a copy to proofread. I guess they could not stand the suspense any longer, which I had carefully cultivated over the last two years. Without going into detail who belonged to which group, I would like to thank Charles Palmer, Jonathan Shapiro, Dan Wallach, Ronald Perez, Paul Karger, and Trent Jaeger who provided me with much useful feedback on my thesis or papers that comprise it.

Besides the people mentioned above, there is the long cast of characters who contributed to my thesis in one way or another. First of all there is the old Amoeba group of which I was part and where I learned many of the traits and got my early ideas. This group, at that time, consisted of Frans Kaashoek, Ed Keizer, Gregory Sharp, Hans van Staveren, Kees Verstoep, and Philip Homburg. Philip has been my office mate at the VU for most of the time and someone with whom I could exchange hacking problems and who provided feedback on part of this thesis. Then there is the ORCA group, consisting, at that time, of Henri Bal, Koen Langendoen, Raoul Bhoedjang, Tim Rühl, and Kees Verstoep. They did a marvelous job at the Orca runtime system which I took advantage of in my implementation.

In the Dutch system, once a thesis is approved by the advisor, it is passed on to a reading committee for a final verdict. My reading committee consisted of Frans Kaashoek, Paul Karger, Sape Mullender, Charles Palmer, and Maarten van Steen. They provided much useful and insightful feedback which greatly improved this thesis.

I also wish to thank the Department of Mathematics and Computer Science of the Vrije Universiteit, N.W.O., Fujitsu Micro Electronics Inc., and IBM T.J. Watson Research Center for providing support for carrying out the research and especially for the generous funding for trips, books, and equipment.

Without a doubt, the hardest part of writing this thesis was the Dutch summary. Mainly because Dutch lacks the appropriate translation for many English computer terms. Fortunately, our Belgium neighbors were very helpful in this respect and put up a web site (in proper Dutch, *een webstek*) with lots of useful suggestions. I have used their site, http://www.elis.rug.ac.be/nederlands.html, frequently.

Finally, but certainly not least I would like to thank Judith for her love and continuing support and of course Sofie for being as quiet as a mouse during the evenings and during the daytime reminding me of her quality time by grabbing the Ethernet cable to my notebook.

Samenvatting

Ontwerp en toepassingen van een uitbreidbaar bedrijfssysteem

Introductie

Traditionale bedrijfssystemen laten zich moeilijk aanpassen aan de eisen van moderne toepassingen. Toepassingen, zoals multimedia, Internet, parallelle, en (in hardware) ingebouwde toepassingen hebben elk erg verschillende bedrijfssysteem eisen. Multimedia toepassingen verwachten dat het bedrijfssysteem bepaalde handelingen op tijd afmaakt, Internet toepassingen verwachten specifieke netwerk-protocol implementaties, parallelle toepassingen verwachten snelle communicatie-primitieven, en ingebouwde toepassingen hebben specifieke geheugeneisen. In dit proefschrift beschrijven we een nieuw bedrijfssysteem, *Paramecium*, dat dynamisch uitbreidbaar is en zich kan aanpassen aan de verschillende eisen van moderne toepassingen. Aan de hand van dit systeem bestuderen we de vraag of uitbreidbare bedrijfssystemen nieuwe toepassingen mogelijk maken die moeilijk of onmogelijk te implementeren zijn in huidige systemen.

Paramecium bestaat uit drie onderdelen: een kern die de basis vormt van het systeem, een aantal systeem extensies die de kern of toepassingen kunnen uitbreiden en een aantal toepassingen die gebruik maken van de kern en de systeem extensies. Alle onderdelen van het systeem zijn uniform gedefinieerd in een objectmodel. Deze uniformiteit zorgt ervoor dat dezelfde extensies zowel in de kern als in de toepassingen kunnen worden geladen, wat een verbetering in de doorlooptijd of een betere bescherming van het systeem kan opleveren.

In deze samenvatting beschrijven we: het objectmodel, de basiskern, een aantal voorbeeld systeem-uitbreidingen en een aantal toepassingen die daar gebruik van maken. Tenslotte geven we een samenvatting van de conclusies van dit proefschrift.

Objectmodel

Het objectmodel is een essentieel onderdeel van Paramecium. Centraal in dit model staan modules, meerdere interfaces per module en een externe naamgeving voor modules en interfaces. Elke module exporteert één of meerdere interfaces die geïmplementeerd worden door de betreffende module. Het voordeel van meerdere interfaces over één interface per module is dat de module op meerdere andere modules kan aansluiten. Bijvoorbeeld, een meerdradig uitvoeringspakket kan een bepaalde interface aanbieden. Als we maar één interface per module zouden hebben, en we willen het systeem uitbreiden met prioriteiten zouden we de interface moeten aanpassen en alle toepassingen die van die interface gebruik maken. Met meerdere interfaces per module kunnen we een nieuwe interface toevoegen zonder de oude interface te veranderen en zo dus het aanpassen van alle toepassingen voorkomen.

Redenen om meerdere interfaces per module te hebben zijn: compatibiliteit, de mogelijkheid om een module te vervangen door een module met dezelfde interface; evolutie, de mogelijkheid om interfaces te evolueren; organisatie, het gebruik van interfaces dwingt een modulaire software organisatie af, wat het overzicht en de onderhoudbaarheid van het systeem ten goede komt.

Elke geïnstantieerde module is geregistreerd in een hiërarchische namenlijst onder een symbolische naam. Om een referenties naar een bepaalde interface te krijgen moet een module deze naam opzoeken in de namenlijst en vervolgens de juiste interface selecteren. Deze namenlijst vormt de basis van de flexibiliteit van Paramecium. Om een service aan te passen is het voldoende om de naam te vervangen door een module met een gelijksoortige interface.

Naast interfaces en een namenlijst, kent het model ook objecten, klasses, en composities. Een object is een aggregatie van data en operaties op die data. Een klasse bevat de implementatie van die operaties waarbij één klasse meerdere object instanties kan hebben. Een compositie is een aggregatie van meerdere objecten waarbij de samenvoeging zich gedraagt als één object. Het objectmodel was ontworpen voor gebruik in Globe en Paramecium. In tegenstelling tot Globe, gebruiken we in Paramecium voornamelijk de module, de namenlijst en de meerdere interfaces per module concepten van het model.

Uitbreidbare Bedrijfsystemen

De belangrijkste richtlijn voor het ontwerp van de basiskern was dat uitsluitend de services die essentieel zijn voor de integriteit van het systeem zijn opgenomen in de basiskern. Alle andere services worden dynamisch geladen naar gelang er behoefte aan is door een toepassing. De kern bevat alleen de primitieve services zoals basisgeheugenbeheer, basisuitvoeringsbeheer, en namenlijstbeheer. Alle andere services, zoals virtueel geheugen, netwerk-protocol implementaties en het meerdradig uitvoeringspakket zijn geïmplementeerd buiten de kern als aparte modules.

De eerste service die door de basiskern wordt aangeboden is geheugenbeheer, waarbij vooral de notie van een context centraal staat. Een *context* is een verzameling van virtuele naar fysieke pagina projecties, een namenlijst, en een fout afhandelingstabel. In tegenstelling tot de meeste andere bedrijfsystemen bevat een context geen executeerbare eenheid. Deze worden verzorgt door een ander mechanisme.

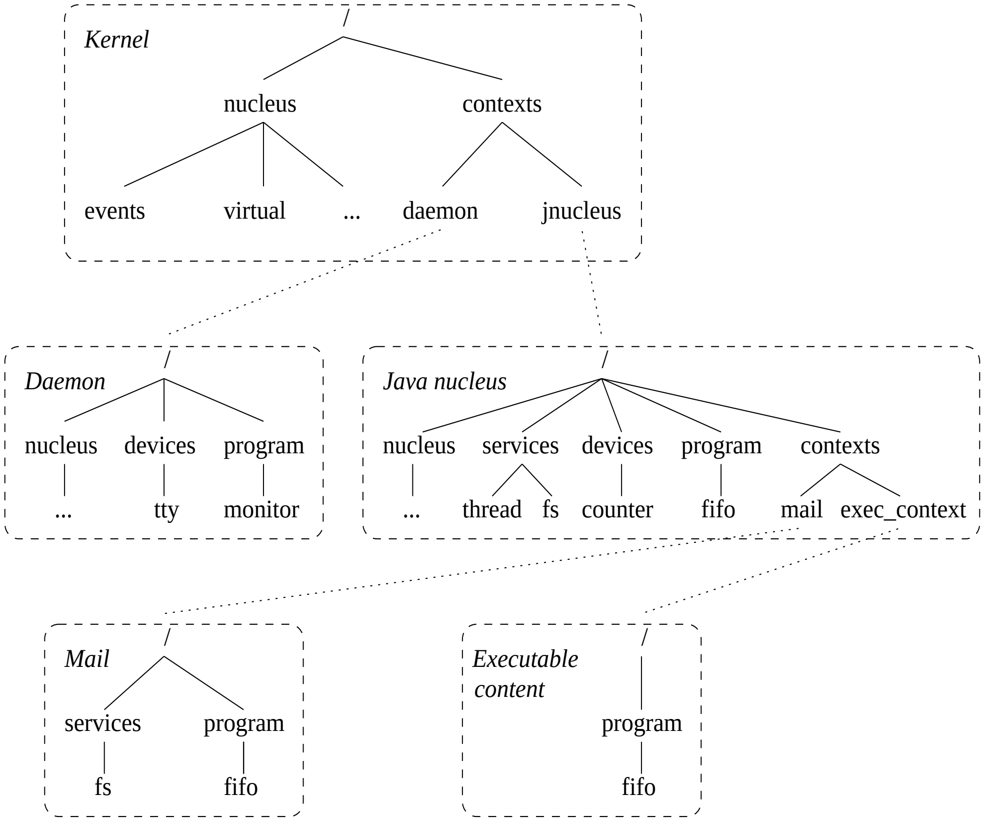
Het geheugenbeheer in Paramecium is onderverdeeld in twee aparte delen, het reserveren van fysiek geheugen en het reserveren van virtueel geheugen. Het is aan de gebruiker van deze primitieven om te bepalen hoe het fysieke geheugen geprojecteerd moet worden op het virtuele geheugen. Traditioneel zijn deze twee gekoppeld, maar het loskoppelen brengt een extra flexibiliteit met zich mee die met name in onze Java virtuele machine gebruikt wordt.

De tweede service bestaat uit uitvoeringsbeheer. Hiervoor gebruikt Paramecium een mechanisme gebaseerd op gebeurtenissen, waarbij asynchrone (*interrupts*) en synchrone (*traps* en explicite invokaties) gebeurtenissen vereenigd zijn in één abstractie. Met elke gebeurtenis zijn een aantal hanteerders geassocieerd, waarbij elke hanteerder uit een routine, een stapel voor de variabelen, en een context bestaat. Als een gebeurtenis gactiveerd wordt, wordt de routine uit de geassocieerde hanteerder onmiddelijk aangeroepen ongeacht wat er op dat moment wordt geexecuteerd. Dit zorgt voor een minimale vertraging tussen het optreden van een gebeurtenis en de afhandeling ervan. Boven op dit mechanisme hebben we traditionele IPC (interproces communicatie) en een meerdradig uitvoeringssysteem gebouwd.

De derde service die de basiskern aanbiedt is het beheren van de namenlijst. Alhoewel elke context een eigen hiërarchische namenlijst heeft, zijn de contexten zelf in een boom structuur georganiseerd, waarbij de kern de wortel van de gerichte graaf is. Een context heeft toegang tot de namenlijst van zichzelf en die van zijn kinderen, maar niet tot die van zijn ouders. Een kern uitbreiding heeft toegang tot de kernnamenlijst en heeft dus toegang tot alle interfaces in het systeem. Voor een voorbeeld van een namenlijst zie Figuur 1.

De basiskern zorgt voor het laden, het registreren en importeren van interfaces en het verwijderen van objecten in het hele systeem. De basiskern zorgt ook voor het instantieëren van schaduw interfaces wanneer een context een interface importeert die niet door die context geïmplementeerd wordt. Er wordt dan automatisch een schaduw interface aangemaakt die de juiste methode in de andere context aanroept.

De belangrijkste eigenschap van de basiskern is dat het dynamisch uitgebreid kan worden met extensies. Daar de belangrijkste functie van de kern het behoud van de integriteit is, dienen extensies aan bepaalde voorwaarden te voldoen. Zij mogen uitsluitend gebruik maken van toegestane geheugen lokaties en uitsluitend die code en externe routines executeren die zijn toegestaan. Het probleem met deze twee veiligeheidseisen is dat ze niet erg formeel zijn. Sommige onderzoekers zijn van mening dat er speciale talen en vertalers nodig zijn om deze veiligheidseisen af te dwingen. Andere onderzoekers gebruiken executie-tijd code-generatie om deze eigenschappen af te dwingen. Wij nemen een ander standpunt in omdat het automatisch afdwingen of verifieëren van deze eigenschappen niet in alle gevallen mogelijk is. Wij zijn van mening dat het uitbreiden van de kern niets anders is dan het uitbreiden van een vertrouwensrelatie waarvoor een digitale handtekening toereikend is. Om een extensie in de basiskern te laden moet de extensie dan ook getekend zijn met de juiste digitale handtekening. Het verkrijgen van deze digitale handtekening is op verschillende manieren mogelijk: door externe validatie, door statische verificatiemethoden of door code-generatie technieken, waarbij er een automatische terugval mogelijkheid is als de gebruikte methode niet toereikend is. Het voordeel van



Figuur 1. Paramecium namenlijst. Elke context heeft zijn eigen namenlijst, hier aangegeven door een gestreept vierkant. De contexten vormen een boom met de kern als wortel.

een digitale handtekening is dat het al deze en andere methodes ondersteunt en ze elkaar kunnen aanvullen.

Systeem Extensies

Naast de kern beschikt Paramecium over een verzameling systeem extensies. Dit zijn services die traditioneel in de kern thuis horen maar die in Paramecium door de toepassing geladen kunnen worden als daar behoefte aan is. In dit proefschrift bespreken we drie van die extensies: een meerdradig uitvoeringspakket, een netwerkprotocol implementatie, en een actief filter mechanisme voor het selecteren en afhandelen van gebeurtenissen.

Het meerdradig uitvoeringspakket voor Paramecium wijkt af van andere systemen in dat het niet in de basiskern zit. De basiskern weet niets van meerdradig uitvoeringen en hoe ze geregeld moeten worden. Daarentegen kent de kern een concept van ketens, een ketting van één of meerdere gebeurtenis aanroepen, waarvoor een co-routine-achtige interface bestaat. Het meerdradig uitvoeringspakket gebruikt dit concept voor zijn implementatie. De andere unieke kenmerken van ons meerdradig uitvoeringspakket zijn dat een uitvoeringen zich van één context naar een andere kan begeven met behoud van dezelfde uitvoeringsidentiteit en dat synchronisatie primitieven op een efficiënte wijze aangeroepen kunnen worden vanuit meerdere contexten.

De tweede systeem extensie is een TCP/IP netwerk-protocol implementatie. Het meest interessante onderdeel hiervan is het bufferbeheer gedeelte dat er voor zorgt dat de verschilende componenten, zoals de netwerk besturingssoftware, de netwerk protocol implementatie en de toepassing, in verschillende contexten geïnstantieerd kunnen zijn zonder dat de data overbodig gekopieërd hoeft te worden. Ons buffersysteem doet dit door slim met fysieke pagina's en virtuele adresruimtes om te gaan.

De derde systeem extensie is een actief filter systeem dat zorgt voor het selecteren en afhandelen van gebeurtenissen. Hier bestuderen we een ander extensie mechanisme waarbij machine onafhankelijkheid essentieel is (in tegenstelling tot de kern extensie mechanismen die we voor de rest van systeem gebruiken). De machine onafhankelijkheid is belangrijk omdat we filter expressies willen migreren naar coprocessoren of andere hardware uitbreidingskaarten. Hiervoor hebben we een eenvoudige virtuele machine gedefinieerd voor filter beschrijvingen. De filters zijn actief omdat ze tijdens een gedeelte van de evalutie neveneffecten mogen hebben. Het is dus mogelijk om eenvoudige bewerkingen geheel in een filter te implementeren.

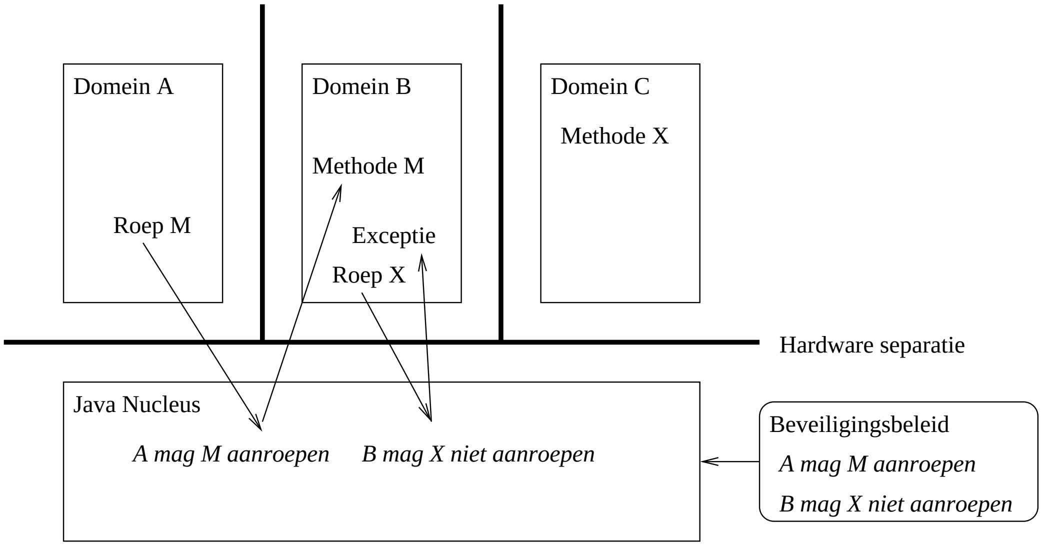
Toepassingen

In dit proefschrift beschrijven we twee toepassingen die we gebouwd hebben boven op de basiskern en bijbehorende systeem extensies. De bedoeling van deze toepassingen is om de flexibiliteit van ons uitbreidbare bedrijfssysteem te valideren. Deze twee toepassingen zijn: een flexibel ondersteuningssysteem voor de parallele programeertaal Orca en een ondersteuningssysteem voor Java wat gebruikt maakt van hardwarematige beveiliging in plaats van softwarematige beveiliging. We beschrijven de twee systemen één voor één.

Ons ondersteuningssysteem voor Orca maakt gebruik van de flexibiliteit van de namenlijst zodat er per Orca object instantie gebruikt gemaakt kan worden van verschillende implementaties. Deze specifieke implementaties kunnen dan een Orca object implementeren met minder strikte ordeningssemantiek en zo de doorlooptijd van het programma verbeteren. Het idee hier achter is dat niet elk gedeeld object een totale ordeningssemantiek vereist zoals dat wordt aangeboden door het standaard ondersteuningssysteem voor Orca.

De tweede toepassing die wij gebouwd hebben is een Java Virtuele Machine (JVM) die Java klasses van elkaar scheidt door gebruik te maken van hardware separatie in plaats van software separatie technieken. Het voordeel van hardware separatie is dat het de beveiligingsmechanismen die het bedrijfssysteem gebruikt hergebruiken en zo de *TCB (trusted computing base)*, dat gedeelte van het systeem waarop de beveiliging rust, aanzienlijk verkleinen en dus de complexiteit van het systeem verminderen. Onze JVM bestaat uit een centraal component, de *Java Nucleus*, die een Java klasse vertaalt naar machine code in een bepaalde context en de beveiligingseisen uit het beveiligingsbeleid afdwingt. In welke context een klasse geplaatst wordt is afhankelijk van een *beveiligingsbeleid* die beschrijft welke klassen in eenzelfde context geplaatst kunnen worden, welke klassen met andere klassen kunnen communiceren, welke objecten gedeeld kunnen worden tussen verschillende contexten en hoeveel geheugenruimte en processortijd een context mag gebruiken.

Wanneer een Java methode wordt aangeroepen die zich in een andere context bevindt, wordt er automatisch een gebeurtenis gegenereerd die wordt afgehandeld door de Java Nucleus. Deze kijkt of, volgens het beveiligingsbeleid, de context daadwerkelijk die methode mag aanroepen. Zo ja, dan wordt de methode in de andere context aangeroepen. Zo nee, dan genereert de Java Nucleus een exceptie (zie Figuur 2). Naast het controleren van methode aanroepen tussen verschillende contexten zorgt de Java Nucleus ook voor het delen van objecten tussen de verschillende contexten. Wanneer tijdens een methode aanroep een object referentie wordt meegegeven zorgt de Java Nucleus ervoor dat dit object beschikbaar is in allebei de contexten inclusief alle objecten waarnaar het refereert.



Figuur 2. De Java Nucleus gebruikt hardware beveiliging om Java klasses te scheiden door ze in aparte contexten te plaatsen. Het beveiligingsbeleid bepaald welke klasse in welke context wordt geplaatst en tot welke methodes het toegang heeft.

De Java Nucleus beheert alle objecten die gebruikt worden door de Java klasses en implementeert één grote adresruimte waar elke context een bepaalde projectie op heeft die afhankelijk is van welke klassen en objecten zich in die context bevinden. Dit is een voorbeeld van toepassingsspecifiek geheugenbeheer dat mogelijk is omdat het reserveren van fysiek en virtueel geheugen gescheiden zijn. Het gebeurtenismechanisme van de kern maakt het mogelijk dat fouten die op individuele virtuele pagina's optreden doorgestuurd kunnen worden naar de Java Nucleus voor verdere afhandeling. Het migrerende meerdradig uitvoeringspakket maakt het verplaatsen van een executeerbare uitvoering tussen contexten erg eenvoudig. De Java Nucleus zelf kan ook gebruikt worden als een kern extensie, dit verbetert de doorlooptijd van de JVM.

Conclusies

De centrale vraagstelling in dit proefschrift was of een uitbreidbaar bedrijfssysteem nuttig was en of het toepassingen mogelijk maakte die moeilijk tot onmogelijk te doen zijn in huidige bedrijfssystemen. Alhoewel deze vraag moeilijk voor alle uitbreidbare bedrijfssysteem te beantwoorden is, kunnen we wel naar de onderzoeksbijdragen van Paramecium specifiek kijken. De belangrijkste onderzoeksbijdragen van dit proefschrift zijn:

- Een eenvoudig objectmodel voor het bouwen van uitbreidbare systemen, waarbij interfaces, objecten, en een externe namenlijst centraal staan.
- Een uitbreidbaar bedrijfssysteem dat gebruikt maakt van een vertrouwensrelatie om de kern uit te breiden.
- Een nieuwe Java Virtuele Machine die gebruikt maakt van hardware fout protectie om Java klasses transparant en efficiënt van elkaar te scheiden.

De resterende onderzoeksbijdragen zijn:

- Een migrerend meerdradig uitvoeringspakket met efficiënte primitieven voor synchronisatie vanuit verschillende contexten.
- Een buffersysteem waarbij de data efficiënt gedeeld kan worden tussen verschillende contexten zonder overbodig te kopieëren.
- Een actief filter mechanisme voor het selecteren en doorgeven van gebeurtenissen.
- Een uitbreidbaar parallel programmeersysteem.
- Een object gebaseerd groep-communicatie-protocol dat gebruik maakt van actieve berichten.
- Een gedetaileerde analyse van IPC en context wisselingen in de kern, de systeem uitbreidingen en de toepassingen.

In dit proefschrift laten we zien dat ons systeem aan aantal toepassingen mogelijk maakt, zoals het efficiënt buffer mechanisme en de Java virtuele machine, die moeilijk te doen zijn in een traditioneel bedrijfssysteem.

1

Introduction

Traditional operating systems tend to get in the way of contemporary application demands. For example, diverse application areas such as continuous media, embedded systems, wide-area communication, and parallel computations all have very different operating system requirements. Providing a single operating system to support all these demands, results in either very large and complex systems that provide the necessary support but at a high cost in added complexity and loss of efficiency, or in application specific and usually very rigid systems.

This thesis is based on the observation that applications have varying operating systems needs and that many applications require more control over their hardware and system services. To investigate this observation, we study the design, implementation, application, and performance of extensible operating systems. These systems enable applications to tailor the operating system by allowing the application to add new or enhance existing services without jeopardizing the security of the system. Programmers can add application-specific customizations to improve performance or add functionality.

Central to this thesis is our own extensible operating system, called *Paramecium*†, which we have designed and implemented. Paramecium is a highly dynamic system, where applications decide at run time which extensions to load, and which allows us to build application-specific or general-purpose operating systems. The foundation of this system is a common software architecture for operating system and application components. We use this architecture to construct a tool box of components. Applications choose their operating system and run-time components from this tool box, or add their own, possibly with application-specific enhancements.

Paramecium's design was driven by the view that the kernel's only essential task is to protect the integrity of the system. Consequently the Paramecium kernel is very small and contains only those resources that are required to preserve this integrity.

†A Paramecium is slightly more advanced than an Amoeba; for example, it knows about sex.

Everything else is loaded on demand either into the kernel or into user space, with the user having control of the placement and implementation. This enables the user to trade off the user-level/kernel-level placement of components that are traditionally found in the kernel, enhance or replace existing services, and even control the memory footprint and real time constraints required for embedded systems.

1.1. Operating Systems

An operating system is a program or set of programs that mediate access to the basic computing resources provided by the underlying hardware. Most operating systems create an environment in which an application can run safely without interference from other applications. In addition, many provide the application with an abstract machine independent interface to the hardware resources that is portable over different platforms.

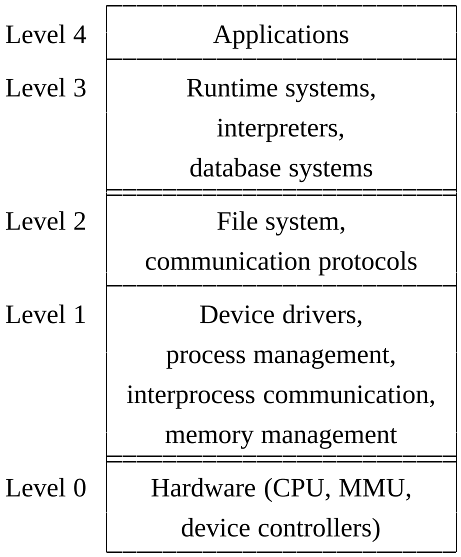
There are two popular views of an operating system: The operating system as a resource manager or the operating system as an abstract virtual machine [Brooks, 1972; Dijkstra, 1968b].

The view of a resource manager has the operating system acting as an arbiter for system resources. These resources may include disks, CD-ROMs, networks, CPU time, *etc*. Resources are shared among various applications depending on each application's requirements, security demands, and priority.

A different view of the operating system is that of an abstract virtual machine. Each virtual machine provides a level of abstraction that hides most of the idiosyncrasies of lower level machines. A virtual machine presents a complete interface to the user of that machine. This principle can be applied recursively.

An operating system provides an interface to its applications to enhance the underlying hardware capabilities. This interface is more portable, provides protection among competing applications, and has a higher level of abstraction than the bare hardware. For example, an operating system can provide a read operation on files rather than on raw disk sectors. Access to these files can be controlled on a per application basis, and the read interface itself is portable over different platforms. A key aspect for many operating systems is to provide fault isolation between concurrent applications. This allows the system to be shared among multiple applications without faults in one application impacting another.

An operating system is composed of various subsystems, each providing a certain service. Following the virtual machine model, a typical operating system consists of the system layers 1, 2, and occasionally layer 3 as depicted in Figure 1.1. The lowest level, 0, consists of the actual computer hardware. This level contains the processor, memory, and peripheral devices. Levels 1 and 2 contain the operating system kernel. Level 1 consists of the core operating system. This core provides process (or protection domain) management, interprocess communication (IPC), memory management, *etc*. Level 2 consist of higher level abstractions, such as file systems for managing



Level 4	Applications
Level 3	Runtime systems, interpreters, database systems
Level 2	File system, communication protocols
Level 1	Device drivers, process management, interprocess communication, memory management
Level 0	Hardware (CPU, MMU, device controllers)

Figure 1.1. Example of a layered system.

storage devices and communication protocols providing reliable cross network communication.

Applications reside outside the operating system kernel. These appear at the highest levels. Here too we can clearly distinguish two layers. Level 3 consists of run-time systems and interpreters (like PVM [Sunderam, 1990] or a Java Virtual Machine [Lindholm and Yelin, 1997]). On top of this resides the application. The application sees the virtual machine provided by the run-time system and the run-time system in turn sees the kernel as its virtual machine.

The actual implementation of an operating system does not have to follow its virtual machine abstractions. In fact, ExOS [Engler *et al.*, 1994] and Paramecium encourage programs to break through the abstraction layers to reduce the performance bottlenecks associated with them.

A number of different architectural organizations are possible for an operating system, ranging from monolithic to a client-server operating system. In the monolithic case, all operating system functions (*i.e.*, levels 1 and 2) are integrated into a single system (*e.g.*, UNIX [Ritchie and Thompson, 1974], and OpenVMS [Kronenberg *et al.*, 1993]). The advantage of this approach is good performance; the disadvantages are that, especially with large systems, it becomes hard to make enhancements to the system because of the complexity. Modular operating systems remedy this problem by isolating subsystems into distinct modules (*e.g.*, Oberon [Wirth and Gütknecht, 1992]). The use of this software engineering technique manages the complexity of the system but can add a performance overhead introduced by the many extra procedure calls. Both monolithic and modular organizations have bad fault isolation properties. For example, it is very hard to contain a memory fault to a single module.

In a client-server operating system approach, the subsystems are implemented as processes that run in their own address space (*e.g.,* QNX [Hildebrand, 1992], Amoeba [Tanenbaum *et al.,* 1991]). The kernel provides rudimentary interprocess communication. This process approach provides much better fault isolation since the subsystems are confined to their own protection domains. The main disadvantage of this approach is the additional performance overhead incurred due to the increased use of interprocess communication. This overhead can easily be more than 1000 times more expensive than a normal procedure call†.

An extensible operating system organization, and especially the system described in this thesis, is a combination of these three architectures. It combines the modular approach and the monolithic approach by allowing modules to be placed into the kernel address space. Modules can also be placed into the user address space providing the advantages of a client-server operating system approach. The placement of modules is determined by a trade-off between fault isolation and performance. Good fault isolation incurs a high cross protection domain call overhead, good performance results in low fault isolation. The main disadvantage of a modular approach is that it precludes macro-optimizations that would have been possible by integrating modules or using the internal information of these modules.

1.2. Extensible Operating Systems

An extensible operating system differs from a traditional operating system in that it consists of a skeletal kernel that can be augmented with specific modules to extend its functionality. More precisely, for the purpose of this thesis we define an extensible operating system in the following way:

> *An extensible operating system is one that is capable of dynamically adding new services or adapting existing ones based on individual application demand without compromising the fault isolation properties of the system.*

Given this definition we can clearly identify the three components that comprise an extensible operating system:

1) *A base system*, which provides a primitive set of services.
2) *Extension mechanisms*, which allows the definition of new services in terms of the basic primitive services.
3) *A collection of modules*, that can be added to the base system using the extension mechanisms.

What is included in the base system differs per extensible operating system. For example, Vino [Seltzer *et al.*, 1996] provides a full monolithic Berkeley UNIX kernel, SPIN [Bershad *et al.*, 1994] provides a microkernel, Paramecium provides a nanoker-

†This is the number for Microsoft's NT 4.0 kernel [Wallach *et al.*, 1997].

nel, and the Exokernel project [Engler *et al.*, 1995] provides a secure way of demultiplexing the underlying hardware. The extension mechanisms also vary greatly among these. SPIN, for example, provides a mechanism to add extensions to each procedure call made in the kernel. The Exokernel provides most of the operating system services in the form of libraries that are part of the user address space and can be replaced or adapted on a per application basis. In Paramecium the kernel provides a small number of primitive functions that cannot be extended; all other services (including thread system, device drivers, *etc*.) are contained in modules which are loaded into the system on demand at run time and can therefore be replaced or adapted.

The most difficult aspect of extending a kernel is maintaining a notion of fault isolation (kernel safety). After all, the kernel prevents different applications from interfering with each other. Adding extra code to the kernel without special precautions can annul the security properties of the kernel. Therefore extensions should be limited and at least provide some form of pointer safety and control safety. Pointer safety prevents an extension from accessing memory outside its assigned regions. Control safety prevents the extension from calling arbitrary procedures and executing privileged instructions.

Enforcing these safety properties can either be done by run-time code generation [Engler *et al.*, 1994], type-safe compilers [Bershad *et al.*, 1995b], proof-carrying code [Necula and Lee, 1996], or code signing. The first two control the code that is inserted into the kernel by sandboxing it. The problem with this method is that unless the extension is severely restricted its safety properties are formally undecidable [Hopcroft and Ullman, 1979]. Proof-carrying code carries with the code a formal proof of its correctness. Generating this proof correctly, however, is not a trivial exercise. The last method, code signing, defers from proving the correctness of the code but rather assigns a notion of trust to it. When the code is trusted enough it is allowed to extend the kernel.

While researchers disagree over the exact nature of the base system and their extension mechanisms, extensible operating systems generally have the following characteristic: they allow safe application specific enhancement to the operating system kernel. This allows them to improve performance by, for example, adding specialized caches, introducing short-cuts, influencing memory allocation, prefetching buffers, *etc*.

Extensible operating systems are also useful for building embedded systems because of their real time potential and control over their memory footprint. For example, in Paramecium it is possible to replace the thread module by one that provides earliest dead-line first (EDF) scheduling rather than round-robin. Controlling the memory footprint is important because embedded systems usually operate under tight time and memory constraints. The ability to adapt the operating system dynamically is especially useful for embedded applications such as personal digital assistant operating systems where the user may run many different applications ranging from interactive games to viewing MPEG movies. A minor, but, for computer scientists, very appealing trait is that extensible operating systems allow easy experimentation with different implementations of operating systems components.

Our definition above emphasizes the ability to extend the system dynamically rather than statically. This explicitly excludes modular operating systems like Choices [Campbell *et al.*, 1987], OSKit [Ford *et al.*, 1997], and Scout [Montz *et al.*, 1994] which are statically configurable, that is, at compile or link time. It also excludes systems such as Solaris [Vahalla, 1996], Linux [Maxwell, 1999], Chorus [Rozier *et al.*, 1988], and Windows NT [Custer, 1993] which provide dynamically loadable kernel modules. The reason for excluding these systems is that in all of them, it is not the application that extends the operating system, but rather a highly specialized and single purpose operating system is constructed for one application. Usually, the application is part of the operating system. Extensible operating systems are more general; they are multipurpose systems that contain specializations for one or more applications. When an application finishes running, the extension is removed from the system.

1.3. Issues in Operating System Research

Some of the research issues in operating system research for the last five years has been dealing with one or more of the following:

- Decomposition and modularization.
- Performance.
- Extensibility.
- Interpretation, and machine independence.

Decomposition and modularization is mainly a software engineering issue. These were considered necessary after the unyielding growth and consequent maintenance problems of monolithic operating systems. The microkernel was deemed the answer to this problem. A small kernel with many separate server processes implementing the operating services. Major proponents of these microkernel systems are: Amoeba, Chorus, Mach [Accetta *et al.*, 1986], Minix [Tanenbaum and Woodhull, 1997], and Spring [Mitchell *et al.*, 1994].

Microkernels did deliver the desired modularization of the system but failed to deliver the performance. This was attributed to two reasons. The first was the large number of cross protection domain calls (IPC) caused by the decomposition into many different server processes. A typical cross protection domain call on a microkernel is of the order of several 100s of microseconds. These add up quickly, especially on multiserver implementations of inherent tightly coupled applications (*e.g.*, the UNIX multiserver implementation [Stevenson and Julin, 1995]).

The second reason for the performance problems is the sharing of large amounts of data across protection domains. This is especially true when sharing network packets, file system or disk buffers. Copy-on-write (introduced by Accent [Fitzgerald and Rashid, 1986] and heavily used by Mach) did not alleviate these problems since it assumes that the data is immutable. When the data is mutable, a separate copy is made on the first write. The multiserver UNIX implementation for Mach showed that large sets of shared data were, in fact, mutable data [Druschel and Peterson, 1992] and copy-on-write failed to deliver the performance hoped for.

Extensible kernels can be considered a retrograde development with respect to microkernels. The microkernel design focuses on moving many services found in monolithic systems outside the kernel address space. Extensible kernels, on the other hand, allow the applications to add specific customizations to the operating systems. When used together with the microkernel concepts, extensibility can be used to reduce the performance penalty introduced by pure microkernel systems and still provide good modularity. For example, a server can extend the kernel with code to handle critical parts of its network processing rather than requiring an expensive cross protection domain call on each incoming network packet.

Separating machine-dependent from machine-independent modules allows modules to be reused among different platforms. Since most modules are platform independent (*e.g.*, file server, process server, time server, *etc*.) this leads to a very portable operating system.

A different approach for reaching this same goal is to provide a platform independent virtual machine as part of the operating system. Examples of these systems are: Forth [Moore and Leach, 1970; Moore, 1974], UCSD P-code [Clark and Koehler, 1982], Inferno [Dorward *et al.*, 1997], JavaOS [Saulpaugh and Mirho, 1999], KaffeOS [Black *et al.*, 2000], and Elate [Tao Systems, 2000]. Each of these platforms runs interpreted code rather than native machine code. The advantage of this approach is that the code can be trivially ported to different architectures. The problem with this approach is that it usually requires on-the-fly compilation techniques to improve the performance. Even then performance is about 30% of that of native code, a penalty not everyone is willing to pay.†

1.4. Paramecium Overview

In this thesis we present a simple extensible system for building application-specific operating systems. We discuss the design, implementation and application of the system. Fundamental to our system is the concept of modules which are loaded dynamically and on demand by the operating system kernel and its applications. An object model defines how these modules are constructed and how they interact with the rest of the system.

Central to the object model are the concepts of multiple interfaces per module and an external interface name space. Modules export one or more interfaces that provide operations which are implemented by that module. The interface references are stored under a symbolic name in a hierarchical name space. The only way to bind to an

†This statement was made by one of the authors of Inferno, David Presotto from Lucent Bell Labs, during a panel session at the Hot Topics in Operating Systems Workshop (HOTOS-IV) in 1997.

interface exported by another module is through this name space. This is the main mechanism through which extensibility is achieved. An operator can replace or override interface names in this name space and refer to different implementations than the system supplied ones.

The Paramecium system itself consists of three parts: kernel, system extensions, and applications. In this thesis we describe examples of each of these. The kernel is a small microkernel. The main design guideline for the kernel was that the base kernel includes only those services that are essential for the integrity of the system. All other services can be added dynamically and on demand by an application. The base kernel includes services such as: memory management, rudimentary thread of control management, and name space management. It excludes services such as demand paging, device drivers, thread packages, and network stacks.

The base kernel implements the concept of an address space, called a *context*, which is essentially a set of virtual to physical memory mappings, a name space, and a fault redirection table. A context does not include a thread of control, these are orthogonal to contexts. The kernel manages physical and virtual memory separately, it is up to the allocator of the memory to create to the virtual to physical mappings.

To manage threads of control, the kernel provides a preemptive *event* mechanism which unifies synchronous and asynchronous traps. This mechanism provides the basis for IPC and interrupt handling. A sequence of consecutive event invocations is called a *chain*. The base kernel provides a co-routine like mechanism to swap among different chains, this is used by our thread package extension to implement thread scheduling.

The base kernel can be augmented or extended by adding new modules to it. To preserve the integrity of the system only appropriately signed modules may be loaded into the kernel. Unlike other systems, the kernel does not enforce control and memory safety. It depends on other mechanisms to guarantee or enforce these properties. The kernel assumes that only modules that conform to these properties are signed. In Paramecium the kernel address space is just another context only with the additional requirement that modules need to be appropriately signed.

The kernel maintains a single global name space for all instantiated module interfaces in the system. Other modules can bind to interfaces stored in this name space, however, their visibility is restricted depending on which context the module is in. A module can only bind to names in the name space belonging to that context or any of its children. It cannot bind to any of its parent’s names. The kernel forms the root of this name space, so any kernel extension can bind to any interface in the system, even when it is in another context. When binding to an interface that is implemented by a module in another context, the kernel automatically instantiates a proxy interface.

On top of this kernel we have implemented a number of system extensions. These extensions are implemented as modules and are instantiated either in the kernel’s context as an extension or in a user’s context as part of its run-time system. The system extensions we describe in this thesis include: a thread package, a TCP/IP stack using a shared buffer implementation, and an efficient filter based demultiplexing service. Our thread package allows a thread of control to migrate over multiple contexts but still behave as a single schedulable entity. The package also provides an efficient mechanism for sharing synchronization state between multiple contexts. Besides passing the thread of control efficiently between multiple contexts, it is equally important to efficiently pass data between contexts without actually copying it. We explore this aspect in our shared buffer implementation which is part of our TCP/IP stack.

Another system extension is our event demultiplexing service. This service dispatches events to interested parties based on the evaluation of one or more filter predicates. Because the requirements for filter predicates are very different and much more restricted than for kernel extensions, we explore a different kind of extensibility: the use of a simplified virtual machine and run-time code generation.

On top of the Paramecium kernel and its system extensions we have implemented two applications: a flexible run-time system for the Orca [Bal *et al.*, 1992] language and a Java virtual machine that uses hardware protection to separate classes. The Orca run-time systems exploits the flexibility features of the object name space to allow programmers to provide specialized implementations for Orca shared objects. These specialized implementations may relax the strict total ordering requirements as dictated by the Orca language and provide weaker semantics for individual objects and therefore reduce the overhead associated with providing stronger semantics.

Our second application, our Java Virtual Machine (JVM), uses many of the features provided by Paramecium to enforce the hardware separation between classes. Unlike traditional JVMs which use software-fault isolation, we have implemented a JVM that uses hardware-fault isolation. This JVM uses run-time code generation and separates classes into multiple contexts. Communication between these classes is handled by a trusted third party, the *Java Nucleus*, which enforces access control on method invocations between different contexts. Besides controlling method invocations, the system also provides an efficient way to share objects and simple data types between different contexts. This mechanism can handle references to other data structures and enforce usage control on the data that is being shared between contexts. For the implementation of our JVM we depend on many concepts provided by the kernel, such as events, contexts, name spaces, separated physical and virtual memory management, and our migrating thread package.

Paramecium is not a paper-only system. Most of the systems described in this thesis have either been completely or partially implemented and run on a Sun (SPARCClassic) workstation. The implemented systems are: the object model support tools, the kernel and support tools, the migratory thread package, the TCP/IP network stack and remote login services, the base Orca run-time system, and the Secure Java Virtual Machine. In addition to these systems, we have also implemented a minimal POSIX support library, a Forth interpreter, a shell and various demonstration and test programs. As it currently stands, Paramecium is not self hosting. We have not fully implemented the active filter mechanism, nor have we implemented enough Orca run-time system extensions to support real applications.

1.5. Thesis Contributions

In this thesis we study the design, implementation, and application of our extensible operating system. With it we try to determine how useful extensible operating systems are and whether they enable new applications that are hard or impossible to do in existing systems.

Fundamental to our approach is the use of a common object model in which all components are expressed [Homburg *et al.*, 1995]. These components are used to construct the operating system at run time. Components can be placed either into the kernel's address space or user address space, while still providing a certain level of protection guarantees [Van Doorn *et al.*, 1995]. To validate this design we have implemented an operating system kernel, a number of components that are typically found in the kernel in traditional systems, and a number of applications. The system components are a thread package and a TCP/IP stack. The applications we have implemented are a simple Orca run-time system and a Java Virtual Machine using hardware fault isolation.

All the components of the system are developed in a specially designed object model. In this model, the operations provided by a component are defined as a set of interfaces. Interfaces are the only way to invoke operations. Other components can import these interfaces. This strict decoupling and emphasis on interface use allows components to be interchanged as long as they export the set of interfaces expected by its user. The object model is used both for Paramecium and Globe [Van Steen *et al.*, 1999]. Globe is an object-based wide-area distributed system intended to replace current *ad hoc* Internet services.

The general idea behind the object model is to provide a tool box of reusable components. These components are loaded on demand by the application. Components are referred to through an external object instance name space. This name space is controlled by the application, enabling it to specify alternate implementations for a specific component.

Our object model is conceptually similar to the ones used in the OSKit [Ford *et al.*, 1997], LavaOS [Jaeger *et al.*, 1998], and Microsoft's Component Object Model (COM) [Microsoft Corporation and Digital Equipment Corporation, 1995]. Each of these projects developed their own object model, even though there is a clearly limited design space, due to different requirements. The OSKit focuses on interfaces for which a subset of COM was found sufficient. LavaOS relies on some of the run time aspects found in CORBA [Otte *et al.*, 1996] and developed a mini-CORBA component model. Paramecium, on the other hand, focuses on interfaces, objects, external object naming, and object compositions. Some of these ideas are novel and some can be traced back to COM and CORBA. For research purposes, we preferred to explore these ideas in a new model rather than limiting ourself by adapting the paradigms of existing models.

Although the object model was designed as an object-oriented system, it turns out that Paramecium puts much more emphasis on its module features, whereas Globe tends to use more of its object features (see Section 2.3 for more details). Both systems, however, rely heavily on the object naming scheme for flexibility and configurability. Globe provides many distributed extensions to the object model that are currently not used by Paramecium.

Rather than using an existing operating system for our experiments we designed our own new system. Our system is highly decomposed into many separate components. The kernel provides a high performance IPC mechanism, user-level device drivers, rudimentary virtual memory support, and digital signature verification for safely down loading extensions. Of the existing promising operating systems at that time (Amoeba, Chorus, Mach, and Spring), none fulfilled our definition of an extensible operating system. That is, they did not provide a minimal base system and extension mechanisms. Modifying existing systems was considered undesirable because their tight integration made it hard to divide the system up into modules.

As Lauer [Lauer, 1981] pointed out, it takes at least 7 years to develop a full featured nontrivial operating system. To reduce this development time we concentrated on building application-specific operating systems [Anderson, 1992]. More specifically, as a proof of concept we concentrated on two applications: an Orca runtime system and a secure Java virtual machine. Both of these systems adapt and extend the operating system in ways that would require a major redesign of contemporary operating systems, if it was even possible to do at all.

For example, Orca is a parallel programming system based on distributed data objects for loosely-coupled systems. Our implementation contains an active message [Von Eicken *et al.*, 1992] component that is integrated into the network device drivers and the thread system. For efficiency, this component bypasses all normal communication protocols. In addition, it is possible to specify alternative implementations per distributed object instance, possibly with different ordering semantics. Integrating this into an existing operating system would require a major redesign.

As a second example we have implemented a secure Java Virtual Machine (JVM). In this example we exploit Paramecium's lightweight protection model where an application is divided into multiple lightweight protection domains that cooperate closely. This provides hardware fault isolation between the various subsystems that comprise an application. As is well-known, the JVM is viewed by many as inherently insecure despite all the efforts to provide it with strong security [Dean *et al.*, 1996; Felten, 1999; Sirer, 1997]. Instead of the traditional software fault isolation based approach, our JVM uses lightweight protection domains to separate Java classes. It also provides access control on cross domain method invocations, enables efficient data sharing between protection domains, and provides memory and CPU resource control. Aside from the performance impact, these measures are all transparent to the Java program if it does not violate the security policy. This protection is transparent, even when a subclass is located in one domain and its superclass is in another. To reduce the performance impact we group classes and share them between protection domains and we map data on demand as it is being shared.

To give an overview, in this thesis we make the following major research contributions:

- A simple object model that combines interfaces, objects, and an object instance naming scheme for building extensible systems.
- An extensible, event-driven operating system that uses digital signatures to extend kernel boundaries while preserving safety guarantees.
- A new Java virtual machine which uses hardware fault isolation to separate Java classes transparently and efficiently.

In addition, we also make the following minor contributions:

- A migrating thread package with efficient cross protection domain synchronization state sharing.
- An efficient cross protection domain shared buffer system.
- An active filter mechanism to support filter based event demultiplexing.
- An extensible parallel programming system.
- An object based group communication mechanism using active messages.
- A detailed analysis of IPC and context switch paths encompassing the kernel, system extensions, and applications.

The common theme underlying these contributions is that the use of an object-based extensible operating system enables useful application specific kernel customizations that are much harder to make or impossible to do in contemporary operating systems.

1.6. Experimental Environment

The experimental environment for this thesis work consisted of a Sun (SPARCClassic) workstation, a SPARC based embedded system board, and a simulator. The Sun workstation was the main development platform. This platform was emulated in software by a behavioral simulator. This class of simulators simulate the behavior of the hardware rather than their precise hardware actions. The simulator and the real hardware both run the same unmodified version of the Paramecium or SunOS operating system. The embedded system board contains a stripped down version of the SPARC processor, a Fujitsu SPARCLite. Its hardware is sufficiently different that it requires a different version of the operating system kernel.

The SPARCClassic we used for our experiments contains a processor implementing the Scalable Process ARChitecture (SPARC) [Sun Microsystems Inc., 1992]. SPARC is an instruction set architecture (ISA) derived from the reduced instruction set computer (RISC) concept [Hennessy *et al.*, 1996]. The SPARC ISA was designed by Sun in 1982 and based on the earlier RISC I and RISC II designs by researchers at the University of California Berkeley [Patterson and Ditzel, 1980].

More specifically, the processor used in the SPARCClassic is a 50 MHz MicroSPARC, a version 8 SPARC implementation. It consists of an integer unit, floating point unit, memory management unit, and a Harvard (separate instruction and data) style cache. The integer unit contains 120 general purpose registers of which only a window of 24 local registers and 8 global registers are visible at any time. Effectively, the register windows contain the top *n* levels of the stack and have to be explicitly saved or restored by the operating system.

Halbert *et. al.* [Halbert and Kessler, 1980] showed that on average the calling depth of a program is five frames. Under this assumption register windows are a good optimization since they eliminate many memory accesses. Unfortunately, some of the underlying assumptions have changed since then, greatly reducing the effectiveness of register windows. These changes were due to the development of object-oriented languages, modularization, and microkernels. Object-oriented languages and modularization tends to trash register window systems because of their much larger calling depth. Microkernels greatly increase the number of protection domain crossings. These are expensive since in traditional systems they require a full register window flush on each protection domain transition and interrupt.

The MicroSPARC has separate instruction and data space caches. Both are one-way set associative (also known as direct mapped) [Handy, 1993]. The instruction cache size is 4 KB and the data cache size is 2 KB.

The memory management unit (MMU) on the SPARCClassic is a standard SPARC reference MMU with a 4 KB page size. Each hardware context can address up to 4 GB. The MMU contains a hardware context table with each entry holding a pointer to the MMU mapping tables. Changing the current hardware context pointer results in changing the virtual memory map. The translation look aside buffer (TLB), a cache of virtual to physical address mappings, is physically tagged and does not have to be flushed on a context switch.

From a processor point of view, the major difference between the embedded system board and the SPARCClassic is the lack of hardware protection and the omission of a floating point unit. Peripherals, such as network, video, and keyboard interfaces that are found on a SPARCClassic are not present on the embedded system board. The board contains two timers, memory refresh logic, and two serial port interfaces (UARTs).

We have implemented our own SPARC Architecture Simulator to aid in the testing of the system and its performance analysis. The simulator is a behavioral simulator along the lines of, although unrelated to, SimOS [Rosenblum *et al.*, 1995]. The simulator is an interpreter and therefore it does not achieve the performance of SimOS. It simulates the hardware of a SPARCClassic in enough detail to boot, run and trace existing operating systems (*e.g.*, Paramecium, Amoeba, and SunOS). Specifically, it simulates a SPARC V8 processor, SPARC reference MMU, MBus/SBus, IO MMU, clocks, UARTs, disks, and Ethernet hardware. Running on a 300 MHz Intel Pentium-II the simulator executes its workload 40-100 times slower than the original machine.

1.7. Thesis Overview

This thesis is organized as follows: The next chapter describes the object model used throughout the Paramecium system. It is the overall structuring mechanism that, to a large extent, enables and defines the degree of extensibility. Objects have one or more named interfaces and are referred to through an external name space. This name space controls the binding process of objects and is constructed by the application programmer or system administrator.

Chapter 3 discusses the Paramecium microkernel. This preemptive event-driven kernel provides the basic object model support, object naming, memory management, and protection domain control. Services that are typically found in the kernel in monolithic systems are components in Paramecium. These components are dynamically loaded at run time, into either the kernel or user address space. Two examples of system components, the thread system and a TCP/IP implementation, are described in Chapter 4.

Two applications of Paramecium are described in Chapter 5. The first application is a run-time system for the parallel programming language Orca where the user can control the individual implementations of shared objects. For example, this can be used to control the ordering semantics of individual shared objects within an Orca program. The second application is a Java virtual machine providing an operating system style protection for Java applets. It achieves this by sandboxing groups of Java classes and instances into separate protection domains and providing efficient cross domain invocations and data sharing.

In Chapter 6 we look a the performance results of the Paramecium system and compare it to other systems. Chapter 7 draws a number of conclusions from our experiments and evaluates in more detail the applications of extensible operating systems.

2

Object Model

In this chapter we study the design and implementation of the object model that is used by Paramecium and Globe [Van Steen *et al.*, 1999] and compare it with other object and component models. The object model provides the foundations for the extensible operating system and applications described in the next chapters. The model consists of two parts: the local object model, which is discussed in this chapter, and the distributed object model which is the topic of a different thesis [Homburg, 2001].

Our object model is a conceptual framework for thinking about problems and their possible solutions. The basis of our model is formed by objects, which are entities that exhibit specific behavior and have attributes. Examples of objects are: files, programs, processes, and devices. Objects provide the means to encapsulate behavior and attributes into a single entity. Objects have operations on them that examine or change these attributes. These operations are grouped into interfaces. In our model, objects can have multiple interfaces. Each interface describes a well-defined set of related operations. An object can be manipulated only by invoking operations from one of its interfaces.

In addition to objects and interfaces, our object model also includes the notion of object composition and object naming. A composite object encapsulates one or more objects into a single object where the result behaves as any other ordinary object. In a way, composite objects are to objects as objects are to behavior and attributes: an encapsulation technique. Object naming provides a name space and the means to bind objects together, manipulate object configurations and aid in the construction of composite objects.

Our object model builds on the following object-oriented concepts [Graham, 1993]:

- *Abstraction* [Liskov *et al.*, 1977] is the concept of grouping related objects and focus on common characteristics. For example, a file is an abstraction of disk blocks and a process is an abstraction of virtual to physical memory mappings, one or more threads of control, file control blocks, permissions, *etc*. Abstractions are used to manage design complexity by allowing the designer to focus on the problem at different levels of detail.

- *Encapsulation* [Parnas, 1972] or *information hiding* is closely related to abstraction and hides implementation details of an object by concentrating on its functionality. As a result encapsulation allows many different implementations that provide the same functionality.
- *Delegation* [Lieberman, 1986] is akin to *inheritance* [Wegner, 1987]. Inheritance allows you to express an object's behavior partially in terms of another object's behavior. Delegation is equivalent to inheritance but allows an object to delegate responsibility to another object rather than inheriting from it. Notice that there is a tension between inheritance, delegation and encapsulation. Encapsulation hides the object instance state while inheritance and delegation mechanisms reveal (part of) the instance state of an object to its subobjects. This is why some authors favor object compositioning over inheritance [Gamma *et al.*, 1995].
- *Polymorphism* [Graham, 1993] is the ability to substitute, at run time, objects with matching interfaces. The mechanism for implementing polymorphism is called *late binding*.

The main advantage of using an object model is that it facilitates the separation of a system into subsystems with well-defined operations on them. This separation into subsystems and defining their interdependencies enables the designer and developer of a system to manage its complexity. The use of composite objects in our model allows the otherwise fine-grained objects to be grouped into coarser-grained objects. A benefit of well-defined independent subsystems and their interfaces is that they have the potential of being reused in different systems.

One of our main goals was to define an object model that is language independent. Hence our focus is on the run time aspects of the model. More formal issues like unique object identifiers or object equality were explicitly left outside of the model.

The main thesis contributions in this chapter are the use of multiple named interfaces per object, the notion of composite objects and the use of an orthogonal name space to provide flexibility and configurability.

2.1. Local Objects

Local objects are used to implement programs or components of programs such as memory allocators, specialized run-time systems, or embedded network protocol stacks. Local objects have multiple interfaces and are named in an object name space. To amortize the cost of the overhead caused by interfaces and object naming, local objects are relatively coarse-grained. For example, an integer is rarely a local object; a thread package implementing many threads of control could be.

Local objects are so called because they are confined to a single address space. Unlike distributed objects, they do not span multiple address spaces, processes, or multiple machines. The reason for this is that distributed objects require much more functionality than local objects. Adding this functionality to every local object would make it very heavyweight (in run-time size and complexity) and most local objects do not need to be distributed. Instead a local object can be turned into a distributed object by placing it in a composition with other objects that implement the desired distribution properties. The remainder of this section discusses local objects; distributed objects are described extensively in Homburg's thesis.

2.1.1. Interfaces

An interface is a collection of methods (or operations) on an object. Most object models associate only one interface per object, which contains all the operations for that object, but in our model we allow multiple interfaces. Each object can export one or more named interfaces and each of these interfaces has a unique name associated with it that describes its methods and semantics. The advantages of multiple interfaces per object are:

- *Plug compatibility.* One of the main reasons to have multiple interfaces is to allow plug compatibility. For example, consider a mail program that requires a network connection to the mail server. The underlying transport does not matter as long as it is stream based. If the mail program uses a generic stream transport interface, any object exporting that interface can be used to provide the transport service whether the transport protocol used is OSI TP4 or TCP/IP. Another example is a thread package providing multiple threads of control. Instrumenting this package with measurements and providing a separate interface to access the measurement data allows the old package to be substituted while its clients are unaware of the change.

- *Interface evolution.* Interfaces tend to evolve over time. For example, when extra methods have to be added or a method signature needs to be changed. With multiple interfaces it is straightforward to provide backward compatibility by providing the old interface in addition to the new interface.

- *Structured organization.* Objects can export many different operations for different functional groups. Multiple interfaces allow these groups to be structured. Consider a random number generator object. The interface containing the operation to obtain random bit strings is very different from the interface that is used by clients that provide the random number object with sources of high entropy. In fact, the user of the random number data is not concerned with the actual fabrication of it. Separating these two related but different functions of the object aids in structuring the system.

The use of interfaces reduces the implementation dependencies between subsystems. Clients should be unaware of the classes of objects they use and their implementation as long as the objects adhere to the interface the client expects. This leads to one of the principles of object-oriented design [Gamma *et al.*, 1995]: *Program to an interface, not an implementation.* Following this guideline allows subsystems to be replaced, adapted or reused independent of their clients.

To support developers in defining their interfaces, we have designed our own interface definition language (IDL) and a generator that takes an IDL description and produces the appropriate definitions for a target language. Currently we support only two target languages, C and C++, and the current IDL reflects that bias. Ideally an IDL contains definitions that are self contained and are target language independent. Recently such a language has been designed and implemented for Globe [Verkaik, 1998], however since it is incompatible with the existing one it is not yet used for Paramecium.

An example of an interface definition in IDL is shown in Figure 2.1 (more extensive examples are given in appendix A). This interface defines three methods alloc, addr, and free. Each method's signature contains a C/C++ like declaration, reflecting its heritage. Each interface has two names associated with it. The internal name, in the example physmem, is used within the IDL to refer to this interface and in the generated output as a type definition. The external name following the equal sign after the last closing bracket, 5 in this example, is the unique interface name. This name is implemented as an extensible bit string and it is used by the client to request this particular interface. This number is the unique name† for an interface and captures its signature and semantics. It is used as a primitive form of type checking and chosen by the developer of the interface.

```
typedef uint64_t resid_t;               // resource identifiers
typedef uint32_t paddr_t;               // physical addresses

interface physmem {
    resid_t    alloc(void);             // allocate one physical page
    paddr_t    addr(resid_t ppage);     // physical address
    void       free(resid_t ppage);     // free page
} = 5;
```

Figure 2.1. Example of an IDL interface definition.

Each interface has associated with it a standard method that is used to obtain another interface. Since it is included in every interface it is implicit in the IDL. This special method has the following type signature:

```
void *get_interface(void *name, int options = 0);
```

†The unique name refers to a world-wide unique name. This is clearly important for the deployment of this model for a large scale distributed system such as Globe.

This method allows the client to access different interfaces from a given interface it already holds. Note that our model does not have the notion of an object handle from which interface pointers can be obtained. That approach would require the programmer to maintain two references to an object: the object handle and the interface pointer. Our approach requires only one reference; this simplifies the bookkeeping for the programmer. The options argument to get_interface allows the client to enumerate all the available interfaces and is mainly used for diagnostic purposes.

Our object model defines a number of standard interfaces. One of them is the *standard object interface*. This interface, shown in Figure 2.2, is supported by all objects and is used to initialize or finalize the object.

Creating a new object is explained fully in the next section and one of the steps involved is to initialize the object. This is done by invoking the init method from the standard object interface. Initialization allows the object implementation to create or precompute its data structures. The initializer is passed as argument its node, called a naming context, in the object name space. The object name space is further explained in Section 2.1.3. This argument is used to locate other objects although the object, by convention, is not allowed to invoke methods on other objects during initialization, other than those contained in the standard object interface. The cleanup method is invoked prior to destroying the object and gives the object a chance to cleanup its own data structures.

Method	**Description**
init(naming_context)	Initialize object
cleanup()	Finalize object

Figure 2.2. Standard object interface.

In addition to the standard object interface a number of other interfaces exist. These interface are optional. One example of such an interface is the persistence interface. This interface allows an object to serialize its data structures and save them on to persistent storage or reconstruct them by reading them from storage. When an object does provide a persistent interface it is invoked after the initialization to reconstruct the data structure or before finalization to save its data structures. The persistent interface is used by the *map* operation.

The run time representation of a standard object interface is shown in Figure 2.3. Each object implementation contains a set of template interfaces out of which the actual interfaces are constructed. The interface templates are identified by their interface name. Each template includes a table with method pointers and their type information. The method pointers refer to the functions implementing the actual methods.

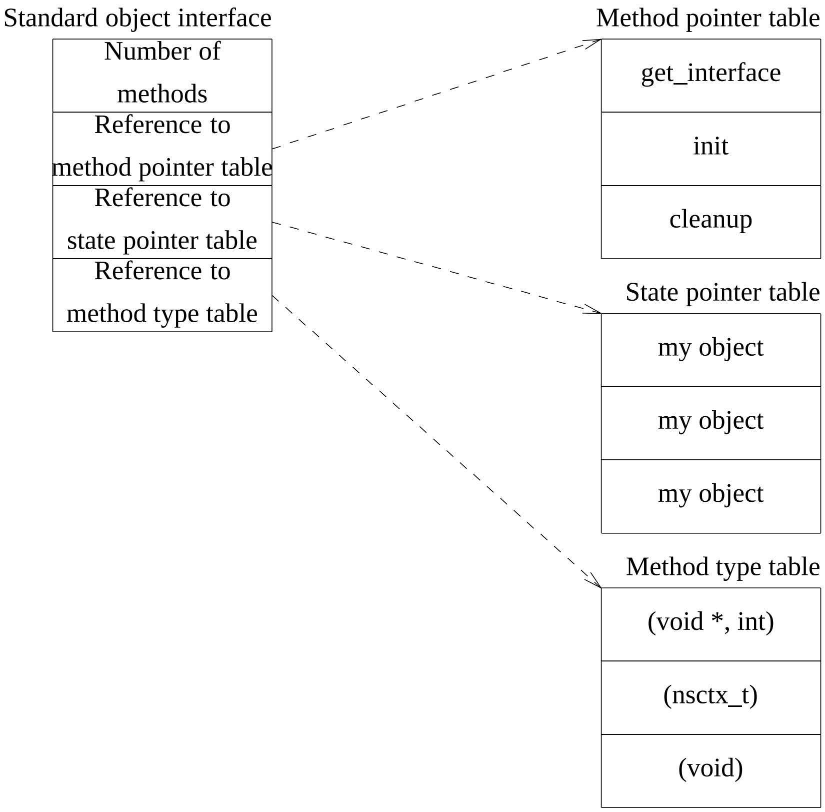


Figure 2.3. Interface implementation and its C definition.

The method type table consists of a rudimentary description of the method's type signature. This information can be used for the dynamic generation of proxy interfaces or to assist symbolic debuggers. Pebble [Gabber *et al.*, 1999], an extensible OS, uses a similar type representation to generate dynamic proxies between processes. In our system, the signature is stored in an array where each element represents the type for the corresponding method argument. Each element contains a type identifier (*i.e.*, 1 for char, 2 for unsigned char, 3 for short, *etc*.), and a size for aggregate types.

The method pointer table and the method type table in the template are shared between all the same interfaces that use the same object implementation. The interface itself consists of pointers to these two tables (as show in the structure definition in Figure 2.3) and a pointer to a table with state pointers. Each method has associated with it its own state pointer; this unusual arrangement is required for composite objects. The state pointer table is unique for each interface.

The IDL compiler generates interface stubs that are used in the programs. For example, for C++ programs the IDL generator maps interfaces onto classes. This allows the programmer to invoke a method by calling a C++ method on the interface that is represented by a C++ class. When a program invokes the init method, the stub will take the current interface pointer and selects from the first slot of the method table the address of the method. This method is then called with the object state pointer contained in the first slot of the state table. This state parameter is followed by optional additional parameters. For C programs the IDL generator maps interfaces onto structures and uses macro definitions to hide the peculiarities of their method invocations.

The method type information serves a number of purposes. Its primary use is to assist the automatic generation of proxy interfaces. Proxy interfaces are used to communicate with interfaces in different address spaces. The type information is also used by debuggers to decode the arguments passed over method invocations and by trace tools to instrument the interfaces.

2.1.2. Objects and Classes

An object is an encapsulation of an instance state and methods operating on that state. In our model, objects are passive; that is, they do not contain one or more embedded threads of control. Objects are also not persistent, although that does not exclude making them persistent through other mechanisms.

The object state is stored in a memory segment that is identified by its base pointer. This pointer is commonly known as the state pointer. All accesses to the object's state are relative to this state pointer. The state pointer is stored in the interface as described in the previous section and is passed implicitly as the first parameter to each method invocation. Decoupling the object state from its methods allows the method implementation to be shared among multiple object instances. In fact, all the methods for each object are contained in a class object specific to that object.

In our model a *class* is a template from which instances of objects are created. A class contains the implementation of the object methods. Consequently each object belongs to a class. A class is a first-class object. That is, the class is itself an object obeying all the rules associated with objects. For instance, a class object is in turn an instance of a class. To break this recursion an artificial super class exists that implements the creating of class instances (that is, map and relocate the implementation in the class).

Each class object provides an interface, called the *standard class interface* (see Figure 2.4). This interface contains methods to create and destroy objects of that class. To create an object the create method of its class is invoked with two parameters, The parameters describe the location of the new object in the object name space. The first parameter contains the directory and the second the symbolic name under which it is registered (see Section 2.1.3 for more details).

The value associated with the registered name is the new object's standard object interface. This interface and the object state are created by the invocation of the class its create method. After the name is registered the object initialization method is invoked. It is up to the object implementation to prevent race conditions resulting from calling other methods before the initialization has completed.

A class may contain variables that are embedded in the class state. These are known as *class variables* and are shared among all objects of that class. They are useful for storing global state such as: list of allocated objects and global measurement

Method	**Description**
interface_soi = create(naming_context, name)	Create a named instance
destroy(interface_soi)	Destroy an instance

Figure 2.4. Standard class interface.

data. Classless objects, also known as modules, combine the implementation of class and object instance into one unit. These modules are useful in situations where there is only one instance of the object; for example: device drivers, memory allocators, or specialized run-time systems.

Our object model does not support class inheritance but does provide a form of delegation. Class inheritance allows one class to inherit properties from another (parent) class. Delegation refers to the delegation of responsibility of performing an operation or finding a value. Wegner [Wegner, 1987] argues that inheritance is a subclass of delegation and that delegation is as powerful as inheritance.

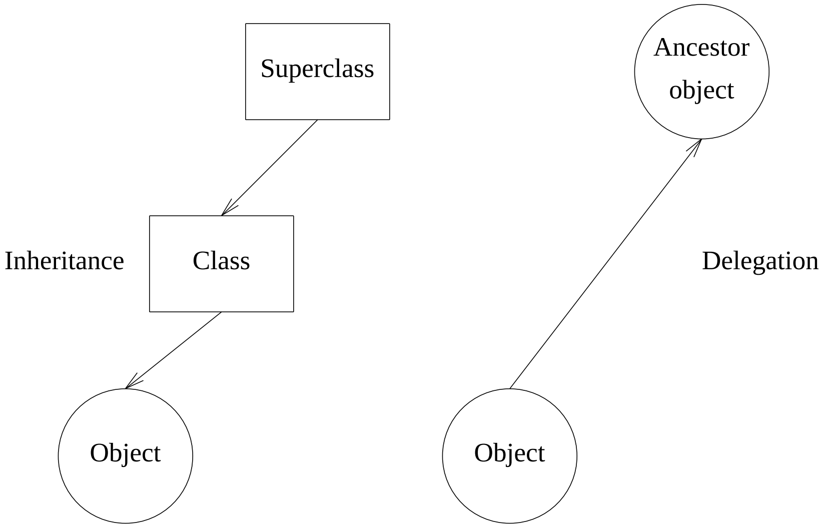


Figure 2.5. Class inheritance *vs.* object delegation.

The difference between inheritance and delegation is shown in Figure 2.5. Key to delegation is that the self reference, *i.e.*, the pointer to the object itself, in an ancestor dynamically refers to the delegating object. The delegated object thereby extends the self identity of the ancestor. In our object model two different kinds of delegation are possible. One where state is shared between the delegating object and one where it is not. In the latter case the delegated method refers to a method and state pointer of the ancestor. When state is shared the ancestor's state is included in the delegating object state and the state pointer in the delegated method refers to this state rather than the ancestor state. This is similar to the implementation of single inheritance [Lippman, 1996]. The disadvantage of this mechanism is that it requires the ancestor to reveal its implementation details to the delegating object.

To aid program development object definitions are expressed in an object definition language (ODL). This language is analogous to the interface definition language. In fact, the IDL is a proper subset of the ODL. The ODL enumerates for each object the interfaces it exports with the names of the routines implementing each method. The ODL generator produces a templates in C or C++ for the listed interfaces and the get_interface methods that are used by the object implementation. The implementor of the object is required to supply only the method implementations.

2.1.3. Object Naming

Run-time object instances can locate each other through a name space. This name space has a hierarchical organization and contains for each object a symbolic name and its object reference. This reference consists of one of the object's interface pointers, usually its standard object interface. The name space is a directed tree where each node is labeled with a symbolic name. Every leaf node has an object reference associated with it. For interior nodes this is optional since these nodes may act as placeholders for the name space hierarchy rather than object instance names.

Each object has a path from the root of the tree to the object itself. Two kinds of path names exist: absolute names and relative names. Absolute names start from the root and name each interior node that is traversed up to and including the object name. The node names are separated by a slash character, ‘‘/’’, and the root is identified by a leading slash character. Relative names, that is those that do not begin with a slash character, start from the current node in the tree. By convention the special name ‘‘..’’ refers to the parent node. This path name mechanism is similar to the naming rules used in UNIX [Ritchie and Thompson, 1974].

An example of a path name is /program/tsp/minimum. This name refers to a shared object that maintains the current minimum for a traveling salesman (TSP) problem. By convention, program refers to a node under which all currently executing programs are located. This node does not have an object associated with it. The tsp node, also an intermediate node, does have an object associated with it representing the TSP program. All objects created by that program are registered relative to its parent node, like the node minimum in this example.

The process of locating an object and obtaining an interface pointer to it is called *binding* and is performed by the *bind* operation. A bind operation always refers to an existing object instance that is already registered in the name space. New objects can be added to the name space either through creating and registering them or loading them from persistent storage. Loading a new object from persistent storage is performed by the *map* operation. For example, the TSP program mentioned above was loaded from a file server using this map operation. Obtaining the measurement data from the minimum object consists of binding to the object and retrieving the data using an appropriate interface.

The example above briefly outlines one of the uses of the object instance name space. The four most important reasons for an explicit object name space that is orthogonal to the objects are:

- *Extensibility.* The name space is the key mechanism for providing extensibility. That is, the ability to dynamically add new or adapt existing services. Since all objects refer to each other through the name space, the operation of changing a name to refer to a different object with similar interfaces replaces an existing service. To facilitate reconfiguration, the name resolution that is part of bind has search rules associated with it. When an object binds to a relative name the name is first resolved using the current node in the tree. If this fails, the name is resolved starting from the parent node. This is applied recursively until the root of the tree is reached. By convention the generic services are registered at the top level of the tree and application specific ones in interior nodes closer to the objects that use them. This scoping allows fine-grained control over the replacement of services.
- *Interpositioning.* Another powerful use of the name space and its search rules is that of object interpositioning [Jones, 1993]. Rather than binding to the object providing the actual service, a client binds to an interposing object that from an interface point of view is indistinguishable from the original object. The interposing object can enhance the service without the service implementation being aware of it. Examples of these enhancements are the addition of cache objects which cache queries and results to the actual service, or trace and measurement objects which keep statistics on the number of invocations to the actual service, or load balance objects that forward the request to the service with the lightest job load, *etc*.
- *Protection.* It is straightforward to extend the name space to multiple trees and confine an object to a single one. If the trees are separated into multiple hardware protection domains it is impossible for an object in one domain to bind to names in another domain. This is the case in the Paramecium kernel. Each protection domain has its own name space in which the services implemented by the kernel are registered. Which services are available depends on a separate security policy that is defined outside the object model; each protection domain starts with an empty name space tree and is populated based on this policy. For example, if the name space does not have an entry for the controlling terminal the objects contained in that protection domain cannot write onto their console. Similar, if the name space holds no file server reference it cannot perform file system operations (see Section 3.4.5 for a more in depth description of protection).

- *Name Resolution Control.* The hierarchical name space structure and search rules help to control and restrict the name resolution which is part of the bind operation. This is especially useful in composite objects where these two mechanisms are used to control the location of objects comprising the composite object.

In addition to the map and bind operations, other operations exist to manipulate the object name space. These are: *register*, *alias*, *delete*, and some additional operations to traverse the name space. The register operation takes a name and an object reference and registers it in the name space. Alias takes two names of which the first name exists in the name space and creates a link to the second name. The delete operation removes a name from the object name space.

The organization of the object name space is largely dictated by conventions, as can be seen in Figure 2.6. This figure shows the name space listing for a single user process running on Paramecium. In this particular instance the process has its own instantiation of a thread package, counter device driver, memory allocator and a shell program. These are all colocated in the same address space.

By convention all program-related objects are stored under the /program node including service objects that are specific to the program. In this example the threads package is stored under /program/services/threads. When the shell program /program/shell binds to services/threads the search rules will first look up the name in the directory in which the shell program is situated. In this case it would resolve to /program/services/threads. If it was not found the system would have tried to resolve the name in the parent directory, in this case the root of tree before announcing a look up failure.

The thread package itself consists of a number of objects of which each implements certain aspects of the thread system. For example, glocal implements per thread data, and sema implements counting semaphores. The interfaces from all these subobjects are exported by the /program/services/threads object. The counter device which provides the timer events that are required for implementing preemptive threads is registered as /program/devices/counter. Here the same search rules apply as before. In order for /program/services/threads to bind to devices/counter the system tries to resolve the name in the /program/services directory. When this fails, the directory /program is used to successfully resolve the name.

As is clear from the example, the hierarchical object name space and the search rules for name resolution introduce a scoping concept of objects that is similar to finding variables in Algol-like languages that allow nested blocks [Backus *et al.*, 1960]. First you start within the closest surrounding block, if it is not found you proceed to the parent block. This scoping enables us to enhance existing services and override the binding to them by placing them in a scope closer to the client of the service. A hierarchical name space with these scoping rules allow fine grained control, per individual object, over the binding process. A single flat name space allows only control

Object names	Description
`/`	Root context
`/program`	Executing program context
`/program/shell`	Shell program
`/program/services`	Services confined to `/program`
`/program/services/threads`	Thread package
`/program/services/threads/thread`	Threads
`/program/services/threads/glocal`	Per thread data
`/program/services/threads/mutex`	Locks
`/program/services/threads/cond`	Condition variables
`/program/services/threads/sema`	Countable semaphores
`/program/services/allocator`	Memory allocator
`/program/devices`	Devices confined to `/program`
`/program/devices/counter`	Counter device driver
`/services`	System wide services
`/services/fs`	File system
`/services/random`	Random number generator
`/nucleus`	Kernel services
`/nucleus/devices`	Device manager
`/nucleus/events`	Event manager
`/nucleus/virtual`	Virtual memory manager
`/nucleus/physical`	Physical memory manager
`/nucleus/chains`	Invocation chains management
`/devices`	System wide devices
`/devices/tty`	Console

Figure 2.6. Object name space hierarchy.

over all object bindings and is therefore less desirable. By convention, absolute names should not be used in bind operation since they prevent the user from controlling the name resolution process.

An example of a system-wide name is /services/fs, which is the object that gives access to the file server. The services exported by the kernel (*i.e.*, system calls) are registered under /nucleus to distinguish them from nonkernel services. These names do not refer to object instances within the process its address space but are links to interfaces in the kernel. The Paramecium kernel detects binds to kernel interfaces and automatically creates proxies for them (see Section 3.4.5). The controlling terminal for this process is registered under /devices/tty and points to an interface from a device driver object.

2.1.4. Object Compositions

Object compositioning is a technique for dynamically encapsulating multiple objects into a single object. The resulting object is called a *composite object*. Externally composite objects exhibit the same properties as primitive objects. That is, they export one or more named interfaces and the client of the object is unaware of its implementation. A composite object is a recipe that describes which objects make up the composition and how they are interconnected. Composite objects are recursive in that a composite object may contain subobjects that are in turn composite objects.

Composite objects are akin to object-oriented frameworks [Deutsch, 1989]. An object-oriented framework defines a set of classes, their interactions, and a description of how the class instances are used together or how they are subclassed. A framework is a concept that defines the architecture of the application and is supported by a subclass hierarchy. Frameworks are a top-down class structuring concept, whereas composite objects are a bottom-up object grouping mechanism with an actual run time realization.

A composite object is also different from an aggregate object in that the subobjects are created dynamically and the binding to subobjects is controlled by the composite. An aggregate object on the other hand usually refers to the notion of combining object types in a static way [Gamma *et al.*, 1995].

The two key issues for composite objects are:

- The construction of external interfaces and the delegation of methods to internal objects.
- Controlling the name resolution within a composite object.

The methods in the interfaces of a composite object are delegated to the methods of its internal objects. Hence the explicit presence of the state pointer in the interface (see Section 2.1.2). This state pointer identifies the internal object state without revealing that state to the user of the composite object. The construction of the composite object and its interfaces occurs through the composite constructor.

```
INIT(composite naming context):
    for obj in subobjects {
        manufacture object obj
        register name of obj in name space
          (relative to composite naming context)
    }
    for obj in subobjects {
        invoke obj.INIT(object’s naming context)
    }
    create external interfaces
```

Figure 2.7. Run-time support for constructing a composite object.

The implementation of a composite object consists of a composite constructor and a list of subobjects that comprise the composite object. The composite constructor, shown in Figure 2.7, first manufactures all the subobjects and registers their instance names into the object name space. Their names, the object naming contexts, are registered relative to the node where the composite object was created. As a result, other subobjects that bind to a name will locate the names in the composition first before finding objects that are implemented outside of it. After manufacturing and registering all the subobjects their individual initialization methods are invoked with as argument their naming context. When a subobject is in turn a composite object, this process is repeated.

How the internal objects are manufactured is left to the constructor implementation: some objects are newly created, some are dynamically loaded from a file server, and some already exist and are bound to. After all subobjects have been created and registered, their individual initialization methods are called. Remember that objects may bind to other objects during this initialization but not invoke methods on them. This prevents race conditions during the initialization stage. To which interfaces (if any) the subobjects bind is left to their implementation. The name space search mechanism guarantees that interfaces to subobjects within the composition are found first, unless the name does not exist within the composition or is an absolute name. When all internal objects are initialized the composite constructor creates all the external interfaces with delegations to the internal object's methods.

To illustrate the use of a composite object consider the example in Figure 2.8. In this, oversimplified because it leaves out locking and revocation, example a client file system object that gives access to a remote file server is extended with a file cache to improve its performance. The resulting file system object with client caching is used by the application instead of the direct access file system object. The new object only caches the actual file content, file meta data, *e.g.*, access control lists, length, modification time, *etc*., is not cached.

Figure 2.8 shows the composite object after it has been manufactured by the composite constructor. The constructor manufactured the file system and cache object by locating the existing file server and binding to it and by creating a new instance of a file cache object. Both these names were registered in the name space directory of the composition so that the cache object could locate the file server object. The cache object exports a file I/O interface which is different from the file system interface and only contains the *open*, *read*, *write*, and *close* methods. The open method prepares the cache for a new file and possibly prefetches some of the data. Read requests are returned from the cache if they are present or requested from the file server and stored in the cache before returning the data. Write operations are stored in the cache and periodically written to the file server. A close flushes unwritten data buffers to the file server and removes the bookkeeping information associated with the file.

After initializing both objects, the constructor creates a new file system interface by combining the interface from the original file system and the interface from the

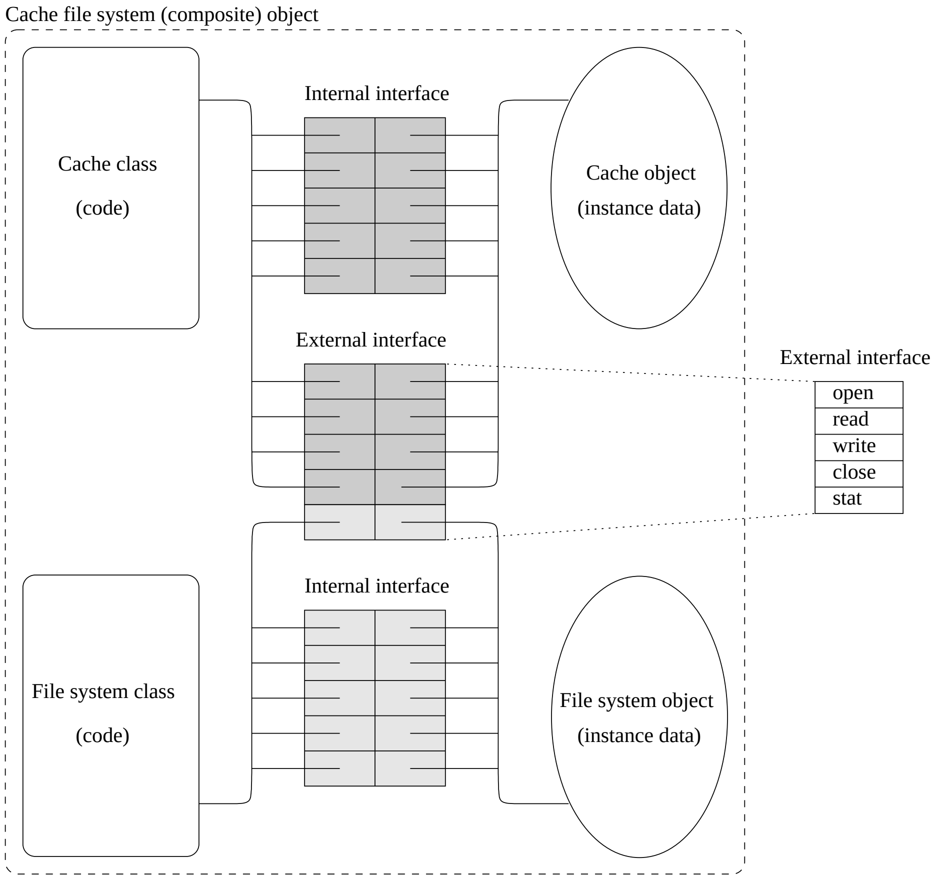


Figure 2.8. Composite object for caching file system.

cache object. The constructor takes the *open*, *read*, *write*, and *close* methods and their state pointers from the cache's file I/O interface and stores them into the new file system interface. The method and state pointer dealing with file meta data, *stat*, is delegated directly to the original file system implementation. It is up to the implementor of the constructor to ensure that when combining interfaces the signature and corresponding semantics of each method are compatible. The constructor may assist by inserting small stub routines that convert or rearrange arguments. It is important that the composite object does not add functionality to the resulting object. Composite objects are only a mechanism for grouping existing objects, controlling their bindings and combining their functionality.

The object names and their interfaces relevant to this composition appear in Figure 2.9. By convention class instances appear under the directory called classes. In this example there are three classes: fs contains the implementation for objects that give access to the remote file server, cfs is the class for the composite object that implements a remote file system client with client caching, and cache implements a caching mechanism. Instances of these classes are registered under the directory called services.

Name	**Description**
`/classes/fs`	File server class
`/classes/cfs`	Cache file server class (composite object)
`/classes/cache`	Cache class
`/services/fs`	File server object
`/services/cfs`	Cached file server object (external interface)
`/services/cfs/fs`	Link to `/services/fs` (internal interface)
`/services/cfs/cache`	Cache object
`/program/browser`	An application
`/program/services/fs`	Link to `/services/cfs`

Figure 2.9. Object name space for caching file system.

Composite objects are a directory in the object name space rather than a leaf. This directory contains the names of the subobjects that make up the composition. These are the names that other subobjects may expect to find. If a subobject is not part of the composition the name search rules apply. The composite constructor manages the interfaces exported by the composite object and attaches them to the composite object name which also doubles as the directory name for all its subobjects. Programs that use the composite object bind to its name instead of the subobject names.

The application is stored in the directory program together with a link to the cache file system. The search rules guarantee that when an application binds to services/fs it will find the extended one by following the link to it. Registering the new file system service under /program makes it the default for the application and all its subobjects. Registering it as /services/fs (*i.e.*, in the root) would make it the default for any objects in the tree.

As is shown in the example above, the internal structure of the composite object is revealed through the object name space. This is a deliberate choice even though it goes against the encapsulation properties of the object. The reason for doing so is to provide assistance for diagnostics and debugging.

2.2. Extensibility

The motivation behind the object model is to provide the foundation for extensible systems. More specifically, it is the foundation for the extensible operating system described in this thesis and the local object support for Globe. The three extension mechanisms provided by our model are:

- Objects as unit of granularity.

- Multiple interfaces per object.
- Controlled late binding using an object name space.

Objects form the units of extensibility. These units can be adapted or replaced by new objects. Multiple interfaces provide us with different ways of interacting with objects enabling plug compatibility and interface evolution. The name space controls the late binding of objects and provides extensibility, interpositioning, protection, and name resolution control.

It is important to realize that although our system is extensible, it is static with respect to the interfaces. That is, anything that cannot be expressed within the scope of the current set of interfaces cannot be extended. For example, interposing an object and providing caching for it only works when the object interface provides enough semantic information to keep the cache coherent. Another example is a thread implementation that does not provide thread priorities. Adding these to the package requires extra methods to set and get the priorities. A client cannot benefit from these modifications unless it is changed to accommodate the new interface.

These limitations are inherent to our extensible system. The extensions are restricted to adapting or enhancing existing services as expressed in the object's interfaces. Adding new services which are not anticipated by the client are beyond the capabilities of our extension mechanism or any other for that matter.

A curious application of extensible systems is that of enhancing binary only systems to specific application needs. This is especially useful for adapting operating systems like Windows NT or UNIX, which are distributed in binary form.

2.3. Discussion and Comparison

The first seminal works on object-oriented programming are probably Dahl and Nygaard on SIMULA 67 [Dahl and Nygaard, 1966], Ingals on Smalltalk 76 [Ingals, 1978], Hewitt on Actor languages [Hewitt, 1976], and Goldberg and Robson on Smalltalk 80 [Goldberg and Robson, 1983]. Object-oriented programming really hit the mainstream with the introduction of C++ [Stroustrup, 1987], a set of extensions on the popular C [Kernighan and Ritchie, 1988] language.

It was soon realized that the object-oriented programming paradigm has a much wider applicability than within a single language and can be used to interconnect different programs or subsystems. These could be written in different languages or even run on different systems. Hence the development of object models such as OMG's CORBA [Otte *et al.*, 1996], Microsoft's object linking and embedding (OLE) which was followed by their Component Object Model (COM) [Microsoft Corporation and Digital Equipment Corporation, 1995] and its networked version Distributed Component Object Model (DCOM).

Our object model has a number of similarities with the models mentioned above. For example, CORBA and our model both use interface definition languages and allow multiple interfaces per object. OLE and our model both share the dynamic loading capabilities which is akin to shared libraries [Arnold, 1986]. Object instance naming and composite objects are unique to our model. The reason for this is that aforementioned object models focus on the distribution of objects between multiple address spaces rather than the local binding and control issues.

Although the interface definition language we use in this thesis does not reflect this, we share the view with COM that the IDL is language independent and only defines a binary interface. This is unlike CORBA, where each Object Request Broker (ORB) vendor provides a language binding for its ORB and supported languages. These bindings provide the marshaling code to transform method calls to the ORB specific calling conventions.

Paramecium is not the only operating system to use an object model. Flux [Ford *et al.*, 1997], an operating system substrate developed at the university of Utah, uses an object model that is a subset of COM. OLE/COM is the object model of choice for Microsoft’s Windows NT [Custer, 1993]. Our reason for developing a new model rather than a subset of an existing one, like Flux does with COM, is that we preferred to explore new ideas rather than being restricted to an existing object model.

Delegation is due to Lieberman [Lieberman, 1986] where it is used as a separate mechanism from inheritance. Wegner [Wegner, 1987] on the other hand views inheritance as a subclass of delegation. Self [Ungar and Smith, 1987] is an object-oriented language strongly influenced by Smalltalk that explored prototypes and delegation as alternatives to classes and inheritance. Our ideas of a classless object model and the use of delegation were inspired by Self.

Even though the object model supports some notion of a class it is largely unused in Paramecium. Instead, much more emphasis is placed on its component and module concepts where in classes and objects are combined. The motivation for this is that in an operating system there are not many instances of coarse-grained objects. For example, there is only one instance of a program, one instance of a thread package, and one instance of the network driver or TCP/IP protocol stack. Operating systems like Flux and Oberon [Wirth and Gütknecht, 1992] also use the module concept rather than the pure object concepts. The reason for introducing a class notion is due to Globe where there are multiple instances of particular classes.

The use of an orthogonal name space to register run-time object instances is novel although somewhat reminiscent of Plan 9’s name space [Pike *et al.*, 1995] and Amoeba’s directory server [Van Renesse, 1989]. These are used to register services on a per process or system basis. Our object naming is per object instance where each instance controls its late binding through search rules in the name space.

Composite objects are a structuring mechanism for combining cooperating objects into an object that is viewed as a single unit by its clients. Related to composite objects are frameworks. These have been used to create a reusable design for applications, and have been applied to system programming in Choices [Campbell *et al.*, 1987; Campbell *et al.*, 1991]. However, both are very different. Composite objects are a bottom-up grouping mechanism while object-oriented frameworks are a top-down class structuring mechanism.

Notes
The object model research presented in this chapter is derived from work done in collaboration with Philip Homburg, Richard Golding, Maarten van Steen, and Wiebren de Jonge. Some of the ideas in this chapter have been presented at the first ASCI conference [Homburg *et al.*, 1995] and at the international workshop on object orientation in operating system (IWOOOS) [Van Steen *et al.*, 1995].

3

Kernel Design for Extensible Systems

The task of an operating system kernel is to provide a certain set of well defined abstractions. These, possibly machine independent, abstractions are used by application programs running on top of the operating system. In most contemporary operating systems these abstractions are fixed and it is impossible to change them or add new ones†. This makes it hard for applications to benefit from advances in hardware or software design that do not fit the existing framework. Consider these examples: user level accessible network hardware does not fit well in existing network protocol stacks; a data base server cannot influence its data placement on a traditional file server to optimize its data accesses; applications that are aware of their virtual memory behavior cannot control their paging strategy accordingly.

This problem appears in both monolithic systems and microkernel systems. In monolithic systems, the size and complexity of the kernel discourages adapting the system to meet the requirements of the application. In fact, one of reasons of the tremendous increase in size of these systems is due to adding functionality specific to many different groups of applications. A typical example of this is UNIX: It started as a small system, but the addition of networking, remote file access, real time scheduling, and virtual memory resulted in a system a hundred fold the size and complexity of the original one. To illustrate the rigidness of these systems, one vendor explicitly prohibits their source license holders from making modifications to the application binary interface.

For microkernel systems the picture is better because most of the services typically found in the kernel for a monolithic system (*e.g.*, file system, virtual memory, networking) reside in separate user processes. But even here it is hard to integrate, for example, user accessible hardware that requires direct memory access (DMA) into the existing framework. The most obvious implementation requires a new kernel abstrac-

†The unavailability of source code for most commercial operating systems only exacerbates this problem.

tion that exports the DMA to a user level application in a safe manner. Other advances might result in adding even more abstractions, this clearly violates the microkernel philosophy.

Another reason to be suspicious of operating systems that provide many different abstractions is their incurred overhead for applications. Applications suffer from loss of available CPU cycles, less available memory, deteriorated cache behavior, and extra protection domain crossings that are used to implement these abstractions. These costs are usually independent of the usage patterns of the applications. For example, applications that do not use the TCP/IP stack still end up wasting CPU cycles because of the amount of continuous background processing required for handling, for example, ICMP, ARP, RARP, and various broadcast protocols.

The problems outlined above all have in common that they require an application to have access to the kernel's internal data structures and to be able to manipulate them. This raises the following key question:

> *What is the appropriate operating system design that exposes kernel information to applications and allows them to modify it in an efficient way?*

This question has been answered by various researchers in different ways, but each evolves around some sort of extensible operating system. That is, a system that enables applications to make specific enhancements to the operating system. These enhancements, for example, can consist of application specific performance improvements or add extra functionality not provided by the original system.

The mechanisms for building these operating systems range from using an ordinary kernel with sophisticated procedure call extension mechanisms to systems that provide raw access to the underlying hardware and expect the application to implement its services. In this chapter we describe our own extensible operating system, *Paramecium*, its design rationale, implementation details, and strengths and, weaknesses.

Paramecium is a highly dynamic nanokernel-like system for building application specific operating systems, in which applications decide at run time which extensions to load. Central to its design is the common software architecture described in the previous chapter, which is used for its operating system and application components. Together these components form a toolbox. The kernel provides some minimal support to dynamically load a component out of this toolbox either in the kernel or a user address space and make it available through a name space. Determining which components reside in user and kernel space is established by the application at execution time.

The main thesis contributions in this chapter are: A simple design for a versatile extensible operating system, a preemptive event driven operating system architecture, a flexible naming scheme enabling easy (re)configuration, a secure dynamic loading scheme for loading user components into the kernel, and a high performance cross domain invocation mechanism for SPARC RISC processors.

3.1. Design Issues and Choices

Paramecium was developed based on our experiences with the kernel for the distributed operating system Amoeba [Tanenbaum *et al.*, 1991]. Paramecium's design was explicitly driven by the following design issues:

- Provide low-latency interprocess communication.
- Separate policy from mechanism.
- Securely extend the kernel's functionality.

Low latency interprocess communication (IPC) between user processes, including low latency interrupt delivery to a user process, was a key design issue. Most contemporary operating systems provide only very high latency IPC. To get an impression of these costs consider Figure 3.1. This table contains the null system and context switch costs for a variety of operating systems (obtained by running *lmbench* [McVoy and Staelin, 1996]). Although not directly related, these two operations do give an impression of the basic IPC cost since an IPC involves trapping into the kernel and switching protection domains. The IPC operation itself only involves a user to supervisor transition, not a context switch because on UNIX the kernel is mapped into each address space.

Ousterhout [Ousterhout, 1990] made an interesting observation in noting that the cost of these operations does not scale very well with the increase in processing power. For example, compared to the SPARCClassic a 275 MHz Alpha is 5.5 times faster but the system call performance is only 3.1 times faster. Ousterhout attributed this to the cost of hardware context switches.

Hardware platform	**Operating system**	**CPU speed (MHz)**	**Null system call (μsec)**	**Context switch (μsec)**
Digital Alpha	OSF1 V3.0	275	12	39
Digital Alpha	OSF1 V3.2	189	15	40
SGI O2	IRIX 6.3	175	9	18
Sun Ultra 1	Solaris 2.5	167	6	16
Intel Pentium	FreeBSD 2.1	133	9	24
Sun SPARCStation 5	Solaris 2.5	110	11	74
Sun SPARCClassic	Solaris 2.6	50	37	168

Figure 3.1. Null system call and context switch latencies.

Low latency IPC is especially important for modular operating systems, such as Amoeba, Mach [Accetta *et al.*, 1986], and LavaOS [Jaeger *et al.*, 1998], that use hardware protection to separate operating system services. As shown in Figure 3.1 the cost of these operations is high while conceptually the operations are simple. For example, a local RPC consists of trapping into the kernel, copying the arguments, switching contexts, and returning from the kernel into the other process. At first glance this does not require many instructions. Hence the question why are these latencies so high?

As Engler [Engler *et al.*, 1995] pointed out, contrary to the high hardware context switch overhead, a major reason for this performance mismatch is the large number of extra operations that must be performed before the actual IPC is executed. These extra operations are the result of abstractions that are tagged onto the basic control transfer. Abstractions like multiple threads of control, priorities, scheduling, virtual memory, *etc*. For example in Amoeba, a system call consists of a context switch, saving the current registers and the MMU context, setting up a new stack and calling the desired routine. When this routine returns, the scheduler is invoked, which checks for queued high-level interrupt handlers (for example clock, network, disk, serial line interrupts). If any of these are queued they are executed first. Then a round-robin scheduling decision is made after which, if it is not blocked, the old registers and the MMU context are restored and the context is switched back to the original user program.

Engler argues that to overcome this performance mismatch, it is important to separate policy from mechanism. Given our Amoeba example above, this comes down to separating the decision to schedule (policy) from the cross protection domain transfer (mechanism). Just concentrating on the pure cross protection domain transfers Engler was able to achieve latencies of 1.4 μsecs on a 25 MHz MIPS processor [Engler *et al.*, 1995].

Furthermore, Engler argues that these policies are induced by abstractions and that there is often a mismatch between the abstractions needed by a program and the ones provided by operating system kernels. For example, an expert system application that wants to implement its own virtual memory page replacement policy is unable to do so in most operating systems since the replacement policy is hardwired into the operating system. Apart from this, existing abstractions are hard to modify and add a performance overhead to applications that do not require it.

For example, applications that do not require network services still end up paying for these abstractions by degraded performance, less available memory, and more complex and therefore error prone systems. Hence, Engler argues that an operating system should provide no abstractions and only provide a secure view of the underlying hardware. Although we disagree with this view (see Section 3.6) we do agree with their observation that the kernel should contain as few abstractions and policies as possible.

We feel, however, that rather than moving services and abstractions *out* of the kernel there is sometimes a legitimate need for moving them *into* the kernel. Examples of this are Amoeba's Bullet file server [Van Renesse *et al.*, 1989], Mach's NORMA RPC [Barrera III, 1991], and the windowing subsystem on Windows NT [Custer, 1993]. These services were moved into the kernel to improve their performance. Ordinarily there are three reasons for moving services or parts of services into the kernel:

- Performance.
- Sharing and arbitration.
- Hardware restrictions.

Performance is the most common reason. For example, the performance of a web server is greatly improved by colocating it in the kernel address space. This eliminates a large number of cross domain calls and memory copies. On the other hand, techniques such as fast IPC [Bershad *et al.*, 1989; Hsieh *et al.*, 1993; Liedtke *et al.*, 1997] and clever buffer management [Pai *et al.*, 2000] greatly reduce these costs for user-level applications and argue in favor of placing services outside of the kernel. However, these fast IPC numbers are deceiving. Future processors, running at gigahertz speeds, will have very long instruction pipe lines and as a result have very high latency context switching times. This impacts the performance of IPC and thread switching and argues again for colocation.

The other reasons for migrating services into the kernel is to take advantage of the kernel's sharing and arbitration facilities. After all, resource management is one of the traditional tasks of an operating system. An example of such a service is IPC redirection [Jaeger *et al.*, 1998]. IPC redirection can be used to implement mandatory access control mechanisms or load balancing.

Finally, the last reason for placing services in the kernel address space is that they have stringent timing constraints or require access to privileged I/O space or privileged instructions that are only available in supervisor mode. In most operating systems only the kernel executes in supervisor mode. Examples of these services are device drivers or thread packages.

Even though we think that there are good reasons for placing specific services in the kernel, as a general rule of thumb services should be placed in separate protection domains for the following reasons: *fault isolation* and *security*. Hardware separation between kernel and user applications isolates faults within the offending protection domain and protects others. From a security perspective, the kernel, the *trusted computing base*, should be kept as minimal as possible. A trusted computing base is the set of all protection mechanisms in a computing system, including hardware, firmware, and software, that together enforce a unified security policy over a product or system [Anderson, 1972; Pfleeger, 1996].

Extending a kernel is not without risk. The extensions should not be malicious or contain programming errors. Since the kernel has access to and manages all other processes, a breach of security inside the kernel is much more devastating than in a user program. The issues involved in extending the kernel securely are discussed extensively in Section 3.3.

The design issues outlined above resulted in the following design choices for Paramecium:

- A modular and configurable system.
- A module can run in either kernel or user mode.
- A kernel extension is signed.
- An event driven architecture.
- Suitable for off the shelf and embedded systems

The most important design choice for Paramecium was *flexibility* and this resulted in a system that is decomposed into many different components that are dynamically configured together to comprise a system. This flexibility is important for building *application-specific operating systems*. That is, operating systems that are specialized for certain tasks. This does not only include traditional embedded systems such as camera or TV microcontrollers, but also general purpose systems such a network computer or a personal digital assistant. Even though the focus of this thesis is on application specific operating systems the techniques and mechanisms can also be used to build a general purpose operating system.

The second design choice was to *safely* extend the kernel by configuring certain components to reside in the kernel. In Paramecium we use code signing to extend the kernels trust relationship and thereby allow user applications to place trusted components into the kernel. In fact, most of Paramecium components have been carefully constructed, unless hardware dictated otherwise, to be independent of their placement in kernel or user space.

The advantage of a decomposed system is that components can be reused among different applications and that the few that require modification can be adapted. Being able to vary the line that separates the kernel from the user application allows certain types of applications, such as the Secure Java Virtual Machine described in Chapter 5, that are hard to implement in existing systems. Furthermore, the ability to vary this line dynamically at run time provides a great platform for experimentation.

In order to achieve *high performance*, Paramecium uses an event driven architecture which unifies asynchronous interrupts and synchronous IPC. This event architecture provides low-latency interrupt delivery by dispatching events directly to user processes. The event mechanism is also the basis for our low-latency IPC mechanism.

In addition to these design choices we adhered to the following principles through the design of Paramecium:

- Nonblocking base kernel.
- Preallocate resources.
- Precompute results.

Given the small number of abstractions supported by the base kernel (*i.e.*, the kernel without extensions) we kept the kernel from blocking. That is, base kernel calls always complete either successfully or with an appropriate error condition. This simplifies the kernel design since it hardly has to keep any continuation state. On the other hand, to allow immediate interrupt delivery, the kernel interfaces are not atomic and require careful handling of method arguments. The second guideline is to preallocate resources and precompute results where appropriate. This principle is used throughout the system to improve the performance. For example, the TCP/IP component preallocates receive buffers and the kernel preallocates interrupt return structures. The main example of a place where precomputation is used is register window handling. The content of various registers, window invalid mask and window invalid mask, are precomputed based on the possible transitions.

The problems outlined in this paragraph are not unique for Paramecium but it does try to solve them in a novel way. Rather than providing a secure view of the hardware Paramecium is an ordinary microkernel albeit one with very few abstractions. Hence it is sometimes referred to as a nanokernel. Paramecium allows code to be inserted into the kernel. Rather than introducing a new concept, such as a special extension language, to express handlers it uses components to extend the kernel. These are reminiscent of kernel loadable modules [Goodheart and Cox, 1994]. Components are loaded on demand by the application and their placement is configurable. Kernel safety is ensured by extending the trust relationship of the operating system rather than using verification techniques. This is further described in Section 3.3. The next section discusses some of the basic kernel abstractions.

3.2. Abstractions

The Paramecium kernel provides a small set of closely related fundamental abstractions. These are contexts, events, and objects. They can be used to implement a full fledged multitasking operating system or a very application specific one. This all depends on the modules loaded into the operating system at configuration time.

A *context* is Paramecium's notion of a protection domain. It combines a virtual-to-physical page mapping together with fault protection and its own object name space. The fault protection includes processor faults (illegal instruction, division by zero) and memory protection faults. A context is used as a firewall to protect different user applications in the system. The user can create new contexts and load executable code into them in the form of objects, which are registered in the context's name space.

In Paramecium the kernel is just another context in the system albeit with some minor differences. The kernel context has access to each context in the system and can execute privileged instructions. Loading executable code into the kernel context, therefore, requires the code to be certified by an external certification authority.

A context is different from the traditional notion of a process in the sense that it lacks an initial thread of control. Instead, a thread of control can enter a context through the event mechanism. Like UNIX, contexts are hierarchical. A parent creates its child contexts, creates event handlers for it, and populates the context's name space. The latter is used by the context to access interfaces to services.

Preemptive events are Paramecium's basic thread of control abstraction. Whenever an event is raised, control is passed to an event handler. This event handler may reside in the current context or in a different context. Context transitions are handled transparently by the event invocation mechanism. Events are raised synchronously or asynchronously. Synchronous events are caused by explicitly raising the event or by processor traps (such as divide by zero, memory faults, *etc*.). Asynchronous events are caused by external interrupts.

Multiple event invocations are called *invocation chains* or *chains*. The kernel supports a coroutine like mechanism for creating, destroying, and swapping invocation chains. Events are the underlying mechanism for cross context interface invocations.

Objects are the containers for executable code and data. They are loaded dynamically on demand into either a user context or the kernel context. In which context they are loaded is under the control of the application, with the provision that only certified objects can be loaded into the kernel context. The kernel implements an extended version of the object name space that is described in Chapter 2. It also supports transparent proxy instantiation for interfaces to objects in other contexts.

Paramecium's protection model is based on capabilities, called *resource identifiers*. Contexts initially start without any capabilities, not even the ability to invoke system calls. It is up to the creator of the context to populate its child with capabilities. The kernel only provides basic protection primitives, it does not enforce a particular protection policy. A good example of this is device allocation. Devices are allocated on a first come first serve basis. A stricter device allocation policy can be enforced by interposing the kernel's device manager with a specialized allocation manager. The allocation manager can implement a policy such that only specified users are allowed to allocate devices. This approach, interposing separate policy managers, is used throughout the system to enforce security.

3.3. Kernel Extension Mechanisms

An operating system kernel enforces the protection within an operating system. It is therefore imperative that kernel extensions, that is code that is inserted into the kernel, exhibit the following two properties:

- *Memory safety*, which refers to memory protection. This requirement guarantees that an extension will not access memory to which it is not authorized. Controlling memory safety is the easiest of the two requirements to fulfill.
- *Control safety*, which refers to the control flow of an extension. This requirement guarantees that an extension will not run code it is not authorized to execute. This means to control which procedures an extension can invoke and the locations it can jump to, and to bound the execution time. In its most general

form, control safety is equivalent to the *halting problem* and is undecidable [Hopcroft and Ullman, 1979].

The remainder of this section discusses the different ways in which researchers ensure memory and control safety properties for kernel extensions. We first discuss the memory safety.

There are three popular ways to ensure memory safety: Software fault isolation [Wahbe *et al.*, 1993], proof-carrying code [Necula and Lee, 1996], and type-safe languages [Bershad *et al.*, 1995a].

Wahbe *et al.* introduced an efficient software-based fault isolation technique (a.k.a. sandboxing), whereby each load, store, and control transfer in an extension is rewritten to include software validity checks. They showed that the resulting extension's performance deteriorated only by 5-30%. It is likely that this slowdown can be further reduced by using compiler optimization techniques that move the validity checks outside of loops, aggregate validity checks, *etc*. In effect, the software-fault isolation technique implements a software MMU, albeit with a 5-30% slowdown.

Necula and Lee introduced a novel concept of proof-carrying code [Necula and Lee, 1996]. It eliminates the slowdown associated with software fault isolation by statically verifying a proof of the extension. This proof asserts that the extension complies with an agreed upon security policy. The extension itself does not contain any extra validity checking code and does not suffer a slowdown. Generating the proof, however, still remains a research topic. Currently the proof is handwritten using an external theorem prover. Ideally the proof is constructed at compile time and associated with the extension.

The third mechanism to ensure memory safety is the use of type-safe languages. This form of protection has a lineage that goes back to, at least, the Burroughs B5000 [Burroughs, 1961]. In current research this protection model is used for kernel extensions in SPIN [Bershad *et al.*, 1995b] and to a certain extent also in ExOS [Engler *et al.*, 1995]. In SPIN an extension is compiled with a trusted compiler that uses type checking, safe language constructs, and, in cases where these two measure fail, validity checks that enforce memory protection at execution time. ExOS uses a slightly different variant whereby it generates the extension code at run time at which point it inserts the validity checks.

Ensuring control safety is a much harder problem, because in its most general form it is equivalent to the *halting problem*. That is, can you write a program that determines whether another program halts on a given input? The answer is no [Hopcroft and Ullman, 1979]. This means that there is no algorithm that can *statically* determine the control safety properties for every possible program. Of course, as with so many theoretical results, enough assumptions can be added to make this proof inapplicable while the result is still practically viable.

In this case, the addition of run-time checks is sufficient to enforce control safety, or a conservative assumption that both execution paths are equally likely. The proof does point out the need for formal underpinnings for methods that claim to provide control safety. This does not only include control safety within a program but also in conjunction with the invocation of external interfaces.

Software fault isolation implements effectively a software MMU and is therefore not control safe. SPIN and ExOS both use trusted compilers or code generators to enforce control safety in addition to run-time safety checks for dynamic branches. Trusted compilers only work when there is enough semantic information available at compile time to enforce control safety for each interface that is used by an extension. For example, part of an interface specification could be that calls to disable_interrupts are followed by calls to enable_interrupts. When the extension omits the later the safety of the extension is clearly violated. Some promising work on providing interface safety guarantees has been done recently by Engler [Engler *et al.*, 2000] but the technique generates false positives and depends on the compiler (or global optimizer) to do a static analysis.

Proof-carrying code asserts that only valid control transfers are made. This does not contradict the nonexistence of the proof described above. That proof shows that there is no general solution. Proof-carrying code asserts that given a security policy the code can only reach a subset of all the states defined by that policy.

To overcome the disadvantages of the extension methods described above (summarized in Figure 3.1), Paramecium uses a different method. Rather than trying to formalize memory and control safety we take the point of view that extending the kernel is essentially extending trust [Van Doorn *et al.*, 1995]. Trust is the basis of computer security and is a partial order relation with a principal at the top. We follow this model closely and introduce a verification authority that is responsible for verifying kernel extensions. The method of verification is left undefined and may include manual code inspections, type-safe compilers, run-time code generation, or any other methods deemed safe. Only those components that are signed by the verification authority can run in the kernel's protection domain. Similar to ours, the SPIN system, which depends on type-safe compilers, uses a digital signature verification mechanism to establish that the extension code was generated by a safe compiler instead of a normal one. Our approach differs from SPIN in that it allows for potentially many different verification techniques.

Rather than formalizing the correctness of the code Paramecium formalizes the trust in the code. This offers a number of advantages over the methods described above. Other than the verification of the signature at load time it does not incur an additional overhead. It provides a framework for many different verification methods that are trusted by the verification authority. Eventually, it is the authority that is responsible for the behavior of the extension. The signature scheme used by Paramecium leaves an audit trail that can be used to locate the responsible verification authority in the event of mishap.

The implementation of this scheme is straightforward given the existence of a public key infrastructure such as X.509 [X.509, 1997]. In the current Paramecium implementation the method is restricted to a single verification authority and therefore

Technique	Memory safety	Control Safety	Remarks
Software fault isolation	Slowdown	Slowdown	5-30% performance degradation for sandboxing
Proof-carrying code	No slowdown, verification at load time	No slowdown, verification at load time	Not practical
Type-safe languages	Slowdown for dynamic memory accesses	Slowdown for dynamic branches	Requires a trusted compiler to compile the module and a mechanism to prevent tampering
Signed code	No slowdown, signature check at load time	No slowdown, signature check at load time	Guarantees trust in the code, not its correctness

Figure 3.2. Extension methods.

uses a unique shared secret per host. This key is only known to the verification authority and the host's TCB. Each component that can be loaded into the kernel address space has a message authentication code (HMAC) [Menezes *et al.*, 1997] in it that covers the binary representation of the component and the shared secret. The component is activated only when the kernel has verified its HMAC. A similar mechanism for signed code is used by Java [Wallach *et al.*, 1997] and the IBM 4758 secure coprocessor [Smith and Weingart, 1999] to verify downloaded code.

During our work with Paramecium, we found that signed modules are not sufficient and that you really need a concept of signed configurations. That is, a signed module can only be trusted to work together with a specified set of other signed modules. Every time an application adds an extension to the kernel it should result in a trusted configuration that has been signed. The reason for this is that even though individual modules are signed and therefore trusted, the semantics expected by one module may not be exactly what is provided by another. For example, our network protocol stack is designed with the assumption that it is the only stack on the system. Instantiating two stacks at the same time will result in competition for the same resources and incorrect behavior for the user of the network protocol stack. Another example is where two applications each instantiate a kernel thread module. Again these two modules are competing for the same resources which may lead to incorrect behavior for the two applications.

A signed configuration mechanism can be used to enforce these additional requirements It can also be used to capture dependencies such as a network device driver depending on the presence of a buffer cache module and a network stack depending on a network device driver. Configuration signatures have not been explored in our current system.

Providing the guarantee of kernel safety is only one aspect of extending a kernel. Other aspects are: late binding and the naming of resources. In Paramecium, modules are dynamically loaded into the kernel and are given an interface to the kernel name service. Using this interface they can obtain interfaces to other kernel or user-level services. In the latter case certain restrictions apply for bulk data transfers. The name space issues are further discussed in Section 3.4.5.

3.4. Paramecium Nucleus

The Paramecium kernel consists of a small number of services and allows extensions to be placed within its address space (see Figure 3.3). Each of these services are essential for the security of the system and cannot be implemented in user space without jeopardizing the integrity of the system. Each service manages one or more system resources. These resources are identified by 64-bit resource identifiers and can be exported to a user address space.

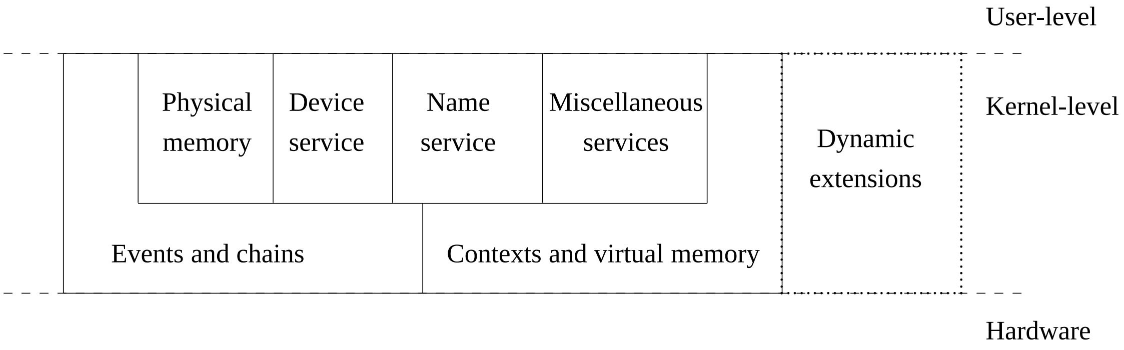


Figure 3.3. Paramecium kernel organization.

The following is a brief overview of the services provided by the kernel and the resources they manage:

- *Context and virtual memory management.* The most central resource provided by the kernel is a context or protection domain. A protection domain is a mapping of virtual to physical addresses, a set of fault events, and a name space. The fault events are raised when a protection domain specific fault occurs, such as division by zero. The name space contains those objects the context can access. Threads are orthogonal to a protection domain and protection domains do not *a priori* include a thread of control. Threads can be created in a protection domain by events that transfer control to it.
- *Physical memory.* Memory management is separated into physical and virtual memory management. The physical memory service allocates physical pages which are then mapped onto a virtual memory address using the virtual

memory service. Physical pages are identified by a generic system-wide resource identifier. Shared memory is implemented by passing this physical page resource identifier to another protection domain and have it map it into its virtual address space.

- *Event and chain management.* The event service implements a preemptive event mechanism. Two kinds of events exist: user defined events, and processor events (synchronous traps and asynchronous interrupts). Each event has associated with it a number of, at least one, handlers. A handler consists of a protection domain identifier, the address of a call-back function, and a stack pointer. Raising an event, either explicitly or through a processor related trap or interrupt, causes control to be transfered to the handler specified by the protection domain identifier and call-back function using the specified handler stack. The event service also has provisions for the handler to determine the identity of the caller domain. This can be used to implement discretionary access control. For each raised event, the kernel keeps some minimal event invocation state containing, for example, the return address. This gives rise to a chain of event invocations when event handlers raise other events. Conceptually, chains are similar to nested procedure calls within a single process. To manage these invocation chains, the kernel provides a primitive set of coroutine like operations to swap and suspend invocation chains.
- *Name service.* Each service exports one or more named interfaces. These interfaces are stored in a hierarchical name space. This name space is managed by the kernel. Each protection domain has a view of its own subtree of the name-space; the kernel address space has a view of the entire tree including all the subtrees of different protection domains.
- *Device allocator.* The Paramecium kernel does not implement device drivers but does arbitrate the allocation of physical devices. Some devices can be shared but most require exclusive access by a single device driver.
- *Miscellaneous services.* A small number of other services are implemented by the kernel. One of these is a random number generator because the kernel has the most sources of entropy.

Each of these services is discussed in more detail below.

3.4.1. Basic Concepts

Each resource (physical page, virtual memory context, event, *etc*.) in Paramecium has a 64-bit capability associated with it which we, for historical reasons, call a resource identifier. As in Amoeba [Tanenbaum *et al.*, 1986], these resource identifiers are sparse random numbers generated by the kernel and managed by user programs. It is important that these resource identifiers are kept secret. Revealing them will grant other user processes access to the resources they stand for.

Resource identifiers are similar to capabilities [Dennis and Van Horn, 1966], and suffer from exactly the same well known confinement problem [Boebert, 1984; Karger and Herbert, 1984; Lampson, 1973]. The possession of a capability grants the holder access to the resource. Given the discretionary access model used by Paramecium, it is impossible to confine capabilities to a protection domain (proof of this is due to Harrison, Ruzzo, and Ullman [Harrison *et al.*, 1976]). Furthermore, the sparse 64-bit number space might prove to be insufficient for adequate protection. Searching this space, assuming that each probe takes 1 nanosecond, for a particular resource takes approximately 292 years on average, this time decreases rapidly when searching for any object when the space is well populated. For example, it takes less than one day to find a valid resource identifier when there are more than $2^{17} \sim 100,000$ resources in the system. From a security point of view this probability is too high.

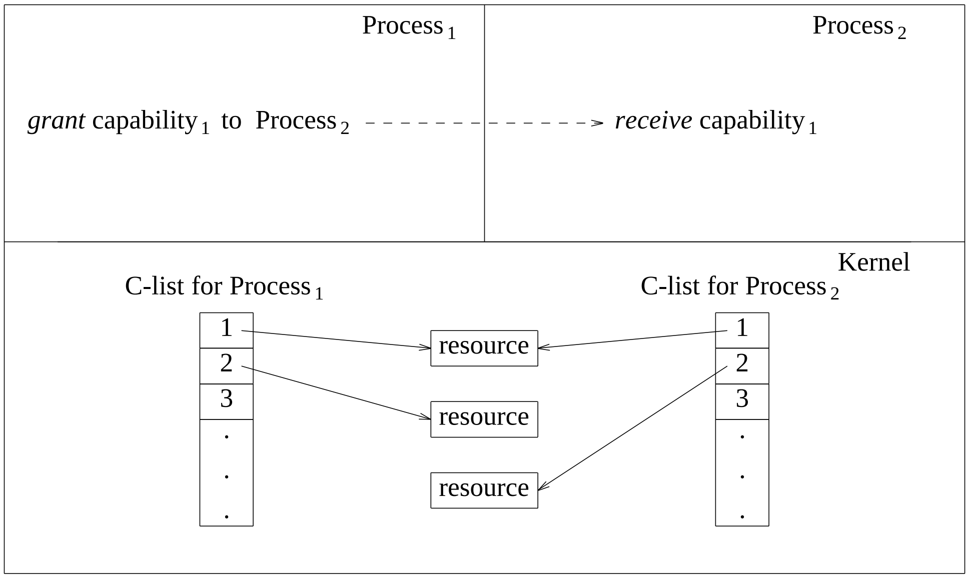


Figure 3.4. Kernel managed resource lists (a.k.a. capability lists). User processes refer to a resource by an index into a per process resource list. Granting another process access to a resource causes the resource to be added to the other process its resource list.

To overcome this critique and allow better confinement of resources, we propose the following modification which is not implemented in the current system. Rather than having the user manage the resource identifiers, they are managed by the kernel. Resources within a protection domain are identified by a descriptor which indexes the resource table in the kernel for that domain (much like a file descriptor table in UNIX). This approach is shown in Figure 3.4 and is similar to Hydra's *capability lists* [Wulf and Harbison, 1981]. In order to pass a resource identifier from one protection domain to another an explicit *grant* operation is required.

The grant operation is implemented by the kernel and takes the resource descriptor, a destination domain, and a permission mask as arguments. The permission mask defines what the destination domain can do with the resource. Currently this only consists of limiting a resource to the destination domain or allowing it to be passed freely. The grant operation returns a resource descriptor in the destination domain if access is granted. This resource descriptor is then communicated to the destination domain through an event or method invocation. Complementary to grant is the *revoke* operation which allows the resource owner to remove access rights for a specified domain.

The implementation described above is similar to the *take-grant* access model [Bishop, 1979; Snyder, 1977] where capabilities have, in addition to the normal read and write rights, also take and grant rights. The read and write rights allows the holder to examine or modify (respectively) the resource associated with the capability. The take and grant rights allow the holder to read and write (respectively) a capability through a given capability (*e.g.*, the take right for a file allows the holder to read capabilities from that file). This model has been extended by Shapiro to *diminish-grant* which provides the ability to obtain capabilities with diminished rights rather than a pure capability take [Shapiro *et al.*, 1999]. Although the take-grant and diminish-grant models do not solve the confinement problem as outlined by Lampson [Lampson, 1973], Shapiro and Weber did show that the diminish-grant model does provide confinement for overt communication channels [Shapiro and Weber, 1997]†. The pure take-grant model does not provide any confinement [Karger, 1988].

Kernel managed resource identifiers have a number of benefits. They can be tightly controlled when shared with other domains. It is also possible for the kernel to intervene and enforce a mandatory access control model onto the shared resources. Finally, resource descriptors consume less memory and register space in that they are smaller than the full 64-bit resource identifier. Of course, within the kernel it is no longer necessary to maintain 64-bit resource identifiers; pointers to the actual resource suffice.

3.4.2. Protection Domains

A protection domain is an abstraction for a unit of protection. It has associated with it a set of access rights to resources and is in most cases physically separated from other protection domains. That is, faults are independent. A fault in one domain does not automatically cause a fault in another. Communication between protection domains is performed by a mediator such as the kernel or a run-time system.

The notion of a protection domain is based on the seminal work on protection by Lampson [Lampson, 1974]. In the Lampson protection model, the world is divided

†Lamspon’s definition [Lampson, 1973] of confinement refers to all communication channels, including covert channels.

into active components, called subjects, passive components called objects†, and policy rules specifying which subjects can access which objects. Together these can be represented in an access control matrix which specifies which subjects can access which objects.

In Lampson's model, a protection domain is a set of rights a process has during its execution, but in operating system research it is often used to describe the fault isolation properties of a process. Hence, in this thesis we use the term *context* to describe an isolated domain which has its own memory mappings, fault handling, and access rights. Note that a context is similar to a process but lacks an implicit thread of control. Threads of control are orthogonal to a context, however, a thread of control is always executing in a context.

Contexts are hierarchical. That is, a parent can create new contexts and has full control over the mappings, fault handling and resources of its children. A default context is empty, it has no memory mappings, no fault handlers, and no accessible resources. For example, a context cannot create other contexts or write to the controlling terminal unless its parent gave it that resource. This notion of an empty context is a basic building block for constructing secure systems.

The advantage of Paramecium contexts over a traditional process based system is that they are much more lightweight. They lack traditional state, such as open file tables, and thread state, and the parent's complete control over a context allows applications to use fine-grained hardware protection. This is especially useful for component based systems, such as COM, that require fine-grained sharing and fault isolation.

At any given time some context is current. Traps occurring during that time are directed using the event table for the current context, as shown in Figure 3.5. Each processor defines a limited number of traps which normally include divide by zero, memory access faults, unaligned memory access, *etc*. Using the event table the appropriate handler is located, which might be in a different context. Device interrupts are not associated with the current context and are stored in a global interrupt event table in the kernel. When an interrupt occurs its event is looked up in the table, after which it is raised. The handler implementing the interrupt handler can reside in a user or kernel context. Unlike context events, interrupt events cannot be set explicitly. Instead they are part of the device manager interface, which is discussed in Section 3.4.6.

The contexts are managed by the kernel, and it exports an interface, see Figure 3.6, with methods to create and destroy contexts and set and clear context fault events (methods create, destroy, setfault, and clrfault respectively). This interface, as well as others in this chapter, are all available to the components inside the kernel as well as to components in user space. The availability of an interface in a user-space context is controlled by a parent context (see Section 3.4.5). Creating a new context

†Objects in the Lampson protection model should not be confused with objects as defined in Chapter 2.

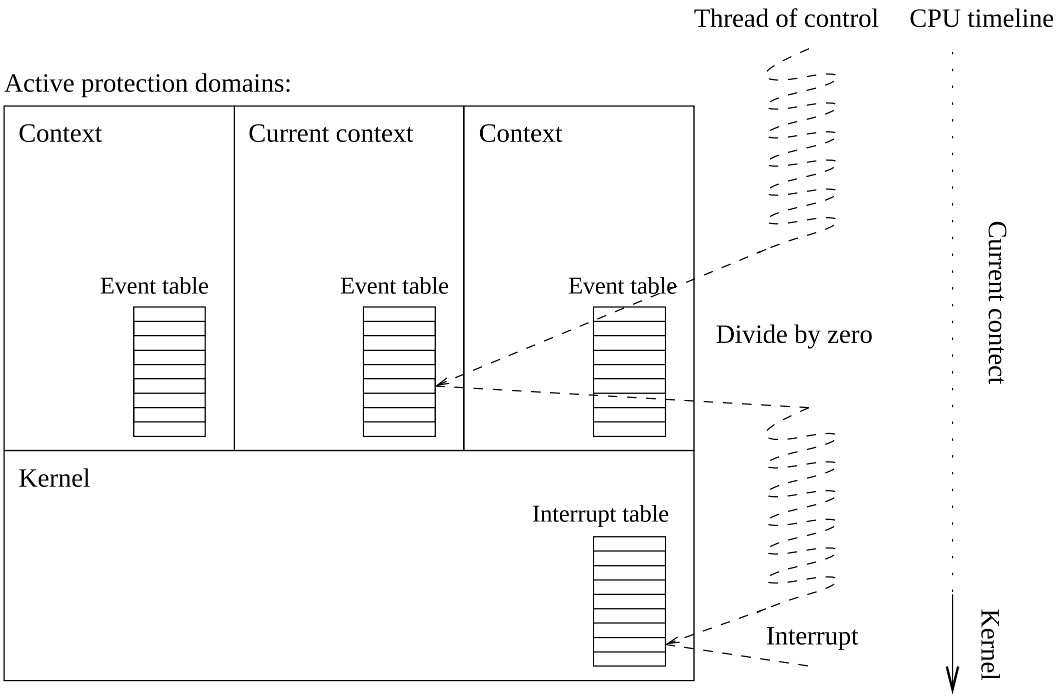


Figure 3.5. Trap versus interrupt handling. Each context has an exception event table, which is kept by the kernel, for handling traps that occur whenever that context is in control. Interrupts are treated differently, these are redirected using a global interrupt event table.

requires a context name and a node in the parent name space which functions as a naming root for child context. This name space is an extension of the object name space in Chapter 2 and is described in Section 3.4.5.

Method	**Description**
context_id = create(name, naming_context)	Create a new context with specified name
destroy(context_id)	Destroy an existing context
setfault(context_id, fault, event_id)	Set event handler for this context
clrfault(context_id, fault)	Clear event handler for this context

Figure 3.6. Context interface.

The context name is generated by the parent, but the kernel ensures that it uniquely names the context by disallowing name collisions. This name is used by the event authentication service, see Section 3.4.7, to identify contexts that raise an event. This identifier is different from the resource identifier returned by the create method.

The latter can be used to destroy contexts, set, and clear fault handlers. The former only serves as a context name to publically identify the context.

3.4.3. Virtual and Physical Memory

One of the most important resources managed by the kernel is memory. In most operating systems memory is organized as a virtual address space that is transparently backed up by physical memory [Daley and Dennis, 1968; Kilburn *et al.*, 1962]. In Paramecium, memory is handled differently. Instead, the application is given low-level control over its own memory mappings and fault handling policy by disassociating virtual from physical memory. The kernel, like most microkernels, only provides an interface for allocating physical memory and creating virtual mappings. Our interface is much simpler than the one used in Mach's virtual memory management system [Rashid *et al.*, 1998] and exposes more low level details about the underlying physical pages. Other services such as demand paging, memory mapped files and shared memory are implemented by the application itself or delegated to an external server. These concepts are similar to Mach's external pagers [Young *et al.*, 1987] and space banks in Eros [Shapiro *et al.*, 1999].

The advantages of separating the two are:

- Reduction of kernel complexity.
- Applications control their own memory policy.
- Easy implementation of shared memory.

By moving the virtual memory management out of the kernel into the application, we greatly reduce the kernel complexity and provide the application with complete control over its memory management. Applications, such as Lisp interpreters, expert systems, or DBMS systems that are able to predict their own paging behavior, can implement their own paging algorithms to improve performance. An example of this is Krueger's work on application-specific virtual memory management [Krueger *et al.*, 1993]. Most applications, however, do not need this flexibility and hand off their memory management to a general-purpose memory server.

Physical memory is managed by the kernel, which keeps unused pages on a free list. Physical memory is allocated using the alloc method from the interface in Figure 3.7. It takes a page from the free list and associates it with the context allocating the memory. A physical page is identified by a resource identifier and on a SPARC its page size is 4 KB. Using the resource identifier the page is mapped into a virtual address space. Sharing memory between two different contexts simply consist of passing the resource identifier for the page to another context. This receiving context has to map it into its own virtual address space before it can be used.

Deallocation of physical memory occurs implicitly when the context is destroyed or when it is explicitly freed using the free method. The physical memory interface also contains a method to determine the physical address of a page, addr. This can be used for implementing cache optimization strategies.

Method	Description
resource_id = alloc()	Allocate one physical page
free(resource_id)	Free a page
physaddr = addr(resource_id)	Return page's physical address

Figure 3.7. Physical memory interface.

The virtual memory interface in Figure 3.8 is considerably more complicated but conceptually similar to the physical memory interface. It too requires explicit allocation and deallocation of virtual memory within a specified context. Besides creating the virtual to physical address mapping, the alloc method also sets the access attributes (*e.g.*, read-only, read-write, execute) and the fault event for the specified virtual page. The fault event is raised whenever a fault condition, such as an access violation or page not present fault occurs on that page. It is up to the fault handler of this event to take adequate action, that is either fix the problem and restart or abort the thread or program. The free method releases the virtual to physical address mapping.

Method	Description
virtaddr = alloc(context_id, hint, access_mode, physpages, event_id)	Allocate virtual address space
free(context_id, virtaddr, size)	Free virtual space
old = attr(context_id, virtaddr, size, attribute)	Set page attributes
resource_id = phys(context_id, virtaddr)	Get physical page resource
resource_id = range(context_id, virtaddr, size)	Get range identifier

Figure 3.8. Virtual memory interface.

Only a program executing in a context or a holder of a context resource identifier is able to control a context's virtual memory mappings using the virtual memory interface. For example, to implement a demand paging memory service, see Figure 3.9, the parent of a newly created context passes the context identifier for it to a separate memory page server. This server interposes the virtual memory interface and examines and modifies each method invocation before passing them on to the kernel. It will record the method arguments in its own internal table and replace the fault handler argument to refer to its own page fault handler. All page faults for this page will end up in the page server rather than the owning context.

When memory gets exhausted, the page server will disassociate a physical page from the context, write the contents to a backing store, and reuse the page for another context. When the owning context refers to the absent page it will cause a page not present fault that is handled by the page server. It will map the page in by obtaining a free page, load it with the original content from the backing store, reinstate the mapping, and return the event which will resume the operation causing the fault. Faults that are not handled by the page server, such as access violations, are passed on to the owning context.

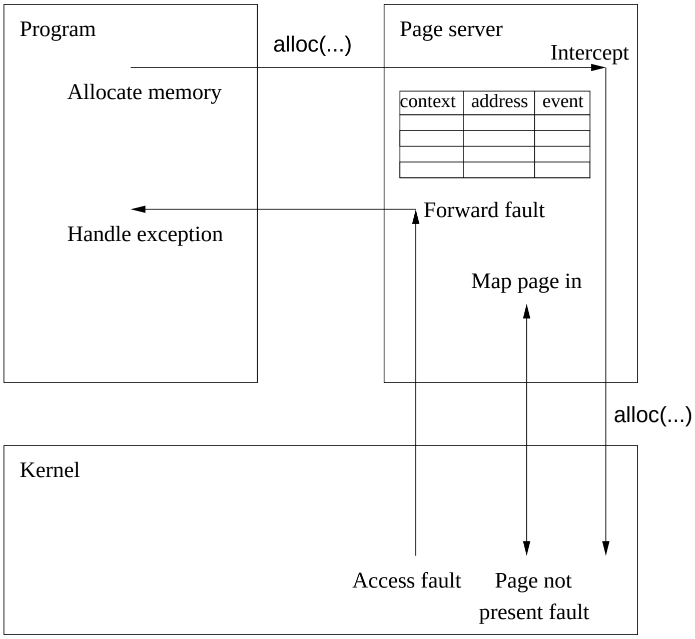


Figure 3.9. Demand paging using a page server.

Of course, this page server only handles memory for contexts that cooperate. A denial of service attacks occurs when a context hogs memory by allocating it directly from the kernel and never returning it. If this is a concern, contexts should never be given access to the actual physical memory interface but to an interposed one that enforces a preset security policy.

The virtual memory interface has three additional methods, attr, phys, and range. The method attr can be used to set and query individual virtual page attributes. Finding the physical page associated with a virtual address is achieved using the phys method.

In some circumstances a context might only want to give away control over a part of its address space. This can be done using the range method. This method creates a resource identifier for a range of virtual memory which the owner can give away. The recipient of that identifier has the same rights for that range as the owner has.

An example of its use is the shared buffer pool service discussed in the next chapter. This service manages multiple memory pools to implement zero-copy buffers among multiple protection domains. Each participating context donates a range of its virtual memory space, usually 16 MB, to the buffer pool service. It then creates the buffers such that the contexts can pass offsets among each other. These offsets give direct access to the buffers which the buffer pool service mapped into each space.

3.4.4. Thread of Control

A thread of control is an abstraction for the execution of a related sequence of instructions. In traditional systems a thread of control is usually associated with a single process, but in more contemporary systems, threads of control can migrate from one protection domain to another. There can also be more than one thread of control. These are either implemented by multiple processors or simulated by multiplexing them on a single processor.

The various ways an operating system can manage these threads of control and their effect on interrupt handling are summarized in figure 3.10. The two traditional methods are event scheduling and event loops.

Thread of control abstraction	**Interrupt handling**	**Preemptable**
Event scheduler	High latency	Yes
Simple event loop	High latency	No
Preemptive events	Low latency	Yes

Figure 3.10. Thread of control management.

Event schedulers, used by, for example, UNIX, Amoeba and Windows NT, schedule events (*i.e.*, interrupts, processes and thread switches, interprocess communication) at well defined points. An event occurs, when a process or thread time slice runs out, when a higher priority process or thread becomes runnable, or when an interrupt has been posted. In these systems external (device) interrupts are handled at two different steps to prevent race conditions. In the first step, the interrupt is acknowledged by the CPU, the CPU state is saved, further interrupts are disabled, and the interrupt handler is called. The first level interrupt handler restores the device state by, for example, copying the data to an internal buffer so it can interrupt again. The handler then informs the scheduler that an interrupt has occurred, restores the CPU state, and resumes the interrupted thread of control. When, eventually, the scheduler is invoked it handles the second level of the interrupt handler. At this point the second level handler is similar to any other thread of control in the system [McKusick *et al.*, 1996].

The main disadvantage of this system is the high interrupt overhead when delivering interrupts to user applications. The application has to wait for or trigger the next scheduler invocation before interrupts are delivered (either as signals or messages). Applications, such as the Orca run-time system, which control the communication hardware in user space need a much lower latency interrupt delivery mechanism.

The second thread of control method is a simple event loop. These are used in Windows, MacOS, PalmOS, Oberon, *etc*. With this method, the main thread of control consists of an event dispatch loop which waits for an event to occur and then calls an operation to handle the event. During the processing of an event other events that occur are queued rather than preempting the current operation. Interrupts are posted and queued as new events. Again, the disadvantage of this system is its high interrupt latency when an event dispatcher is not waiting for an event.

Hybrid thread of control mechanisms are also possible. Examples of these are message passing systems where the application consists of an event loop which accepts messages and the kernel uses an event scheduler to schedule different processes or threads. Examples of these systems are Amoeba and Eros.

Paramecium uses a slightly different form of thread of control management: *preemptive events*. The basic control method is that of an event loop but new events preempt the current operation immediately rather than being queued. The main advantage of this method is the low latency interrupt delivery. When an interrupt occurs control is immediately transfered to the handler of the respective event, even when the thread of control is executing system code. The obvious drawback is that the programmer has to handle the concurrency caused by this preemption. Most of these concurrency problems are handled by the thread package described in the next chapter.

The remainder of this section discusses the design rationale for our integrated event mechanism, the kernel interfaces, and the specifics of an efficient implementation on the SPARC architecture.

Events

To enable efficient user-level interrupt handling Paramecium uses a preemptive event mechanism to dispatch interrupts to user-level programs. Rather than introducing separate mechanisms for handling user-level interrupts and interprocess communication (IPC) we chose to integrate the two mechanisms into a single integrated event mechanism. The advantages of integrating these mechanisms are a single unified communication abstraction and a reduction of implementation complexity. The main motivation behind an integrated event scheme was our experience with the Amoeba operating system. This system supports three separate communication mechanisms, asynchronous signals, RPC, and group communication. Each of these have different semantics and interfaces, and using combinations of them in a single application requires careful handling by the programmer [Kaashoek, 1992].

Paramecium's unified event mechanism combines the following three kinds of events:

- Synchronous interrupts and processor faults such as divide by zero, instruction access violations, or invalid address faults. These traps are caused by exceptions in the software running on the processor.
- Asynchronous interrupts. These interrupts are caused by external devices.
- Explicit event invocations by the software running on the processor.

Each event has a handler associated with it that is executed when the event is raised. An event handler consists of a function pointer, a stack pointer and a context identifier. The function pointer is the address of the handler routine that is executed on an event invocation. This function executes in the protection domain identified by the context identifier and uses the specified stack for its automatic variable storage and activation records. During the executing of its handler the event handler is blocked to prevent overwriting the stack, the single nonsharable resource. That is, event handlers are not reentrant. To allow concurrent event invocations, each event can have more than one handler. These are activated as soon as the event occurs. Invocations of events do not queue or block when there are no handlers available, instead the invocation returns an error indicating that the invocation needs to be retried. When an event handler finishes execution, it is made available for the next event occurrence.

Raising an event causes the current thread of control to transfer to one of the event's handlers. This handler may be implemented in a protection domain different from the current thread of control context. When such a handler is invoked, the current thread of control will be transfered to the appropriate protection domain. This effectively creates a chain of event handler invocations. Such a chain is called an *event* (*invocation*) *chain* and is maintained by the kernel. To manage these event chains, the kernel provides a coroutine like interface to create, destroy, and swap different chains.

An example of a chain is shown in Figure 3.11. Here, a thread in context A invokes an event for which the handler resides in context B. This results in a transfer of control to context B (step 1 and 2). Similar, in context B that thread executes a branch operation, a kind of invocation see below, which causes control to be transfered to context C (step 3 and 4). Although the thread of control passed three different contexts it is still part of the same logical entity, its chain.

Chains provide an efficient mechanism to transfer control from one context to another without changing the schedulable entity of the thread of control. It is the underlying mechanism for our migrating thread package which is described in Section 4.1. The motivation behind the chains abstraction is to provide a fast cross context transfer mechanism that fits in seamlessly with the event mechanism and that does not require the kernel to block such as with *rendez-vous* [Barnes, 1989], operations. In addition, Ford and Lepreau [Ford and Lepreau, 1994] have shown that migrating threads, an abstraction that is very similar to chains, improved the interprocess communication latency on their system by a factor of 1.7 to 3.4 over normal local RPC.

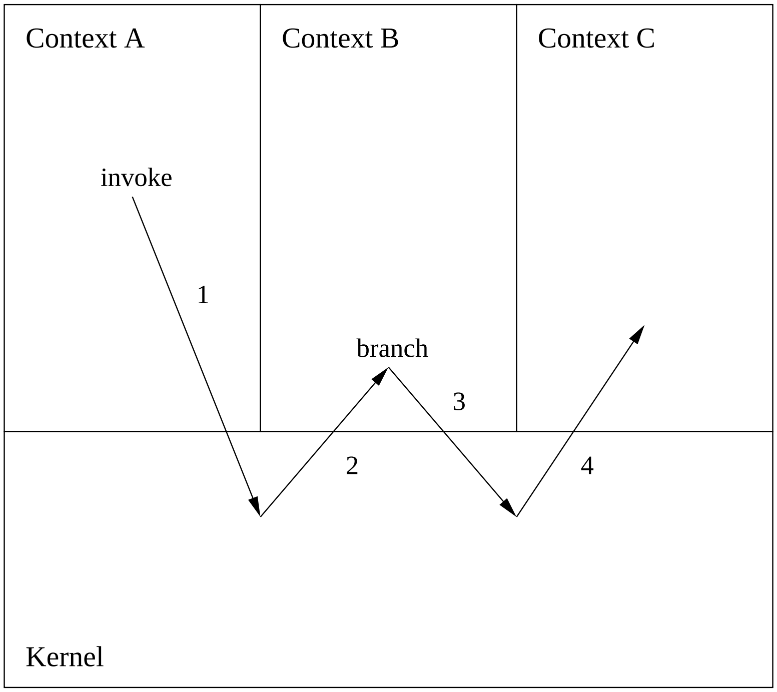


Figure 3.11. Example of an event invocation chain.

The reason for this is that traditional interprocess communication mechanisms, such as a mailbox [Accetta *et al.*, 1986] or *rendez-vous* have a considerable overhead because they involve many extra context switches.

Unlike most contemporary operating system (such as LavaOS [Jaeger *et al.*, 1998], Amoeba, and SPACE [Probert *et al.*, 1991]), Paramecium does not provide threads as one of its basic kernel abstractions. Instead it provides the above mentioned event chains. The motivation behind this is that inherent to a thread implementation are a large number of policy decisions, these include thread priorities, thread scheduling (round robin, earliest dead-line first, *etc*.), synchronization primitives, and locking strategy. These vary per application. Therefore, the thread component is not a fixed kernel abstraction but a dynamic loadable object. For its implementation it uses the event chain abstraction.

In the remainder of this section we describe the two different kind of events, synchronous and asynchronous events, in greater detail including their implementation details for a SPARC RISC processor.

Synchronous Events

Synchronous events are event invocations that are caused by:

1) *Explicit event invocations*. These are caused by calling the invoke or branch methods of the event interface (see below).

2) *Processor traps*. These are caused by the execution of special instructions. such as the SPARC trap instruction, ta, or breakpoint traps.

3) *Synchronous faults*. These are caused by, for example, illegal instruction traps, memory violations, and bus errors.

Each event has one or more handlers associated with it. A synchronous event causes control to be transfered to the first handler that is available. The activation of a handler is similar to a local RPC call [Bershad *et al.*, 1989], in that it passes control to the specified context and continues execution at the program counter using the handler stack. In case of an explicit event invocation additional arguments are copied onto the handler stack. When the handler returns the invocation returns as well.

Upon an event invocation, the first handler is taken from the event inactive handler list and marked active. When there are no event handlers left, a fall back event is generated to signal an exception. This exception implements an application specific mechanism, for example a time out, to restart the invocation. When there are no fall back handlers left, the faulting context is destroyed. This exception handling mechanism is the result of the explicit decision to leave the scheduler (policy) outside the kernel and to allow application specific handling of invocation failures.

An event invocation chain is a sequence of active event invocations made by the same thread of control. As with all other kernel resources, a chain is identified by a 64 bit resource id. When creating a chain the caller has to provide a function where execution is supposed to start, a stack and optional arguments which are passed to the function. The chain abstraction is the basis for the thread management system that provides scheduling. Event invocations cause the chain to extend to possibly different contexts. Even though the chain is executing in another context it can still be managed by the invoking context by using its resource identifier. The invocation chain is maintained in the kernel and consists of a list of return information structures. These structures contain the machine state (registers, MMU context, *etc*.) necessary to resume from an event invocation.

Raising an event can be done in two different ways. The first is a *call*, which is a straightforward invocation where control is returned to the invokee after the handler has finished. The second is a *branch* (see Figure 3.12). A branch is similar to an invocation except that it does not return to the current invokee but to the previous one. That is, it skips a level of event activation. The advantage of a branch is that the resources held by the current event invocation, *i.e.*, event handler stack and kernel data structures, are relinquished before executing the branch. For example, consider an application calling the invoke method in the kernel which then invokes an event handler in a different context. Upon return of this handler control is transfered back to the kernel. At that point resources held by the kernel are released and control is returned to the application. A more efficient implementation uses the branch mechanism. Here the kernel uses a branch method, which releases the kernel held resources, before invoking the event handler. When this handler returns, control is passed back to the application rather than to the kernel.

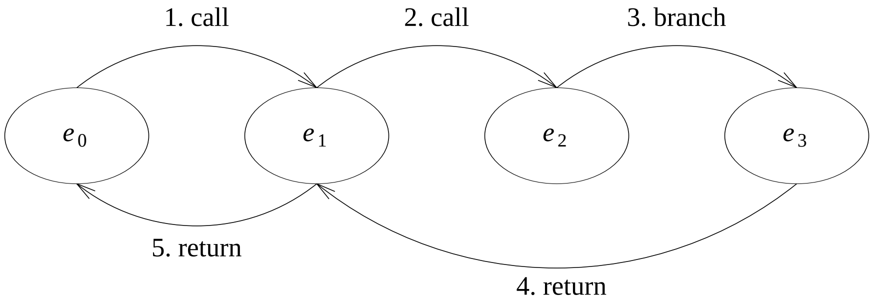


Figure 3.12. Synchronous event invocation primitives.

Once a handler is executing, it cannot be reinvoked since its stack is in use. That is, handlers are not reentrant. A special operation exists to detach the current stack from the event handler and replace it with a new stack. The event handler with its new stack is then placed on the inactive handler list ready to be reactivated. The old stack, on which the thread of control is still executing can then be associated with a newly created thread. This detach ability is used to implement pop-up threads.

Under certain circumstances it is important to separate the authorization to create and delete an event from registering a new handler. For example, adding a handler to an event is different from deleting that event or raising it. In order to accomplish this, we use a dual naming scheme. Each event has a public event name and a private event identifier. The event name is used to register new handlers. Deleting an event, however, requires possession of a valid event identifier.

Asynchronous events

Paramecium unifies synchronous and asynchronous events into a single mechanism. It turns asynchronous events, that is device interrupts, into synchronous event invocations that preempt the current thread of control. The immediate invocation of an event handler provides low latency interrupt delivery to a user level process.

When an interrupt occurs, the current chain is interrupted and an event invocation representing the interrupt is pushed onto the current chain. This invocation causes an interrupt handler to run and when the handler returns the original chain is resumed. Since the interrupt preempts an ongoing operation its handler needs to be short or it has to promote itself to a thread (see Chapter 4). Unfortunately, this simple interrupt mechanism is obscured by interrupt priority levels.

Most processors have multiple devices each capable of interrupting the main processors (*e.g.*, network, SCSI, UART devices). Each of these devices is given an interrupt priority where a higher priority takes precedence over a lower priority. For example, normally a SPARC processor executes at priority 0. A level 1 interrupt will preempt the operation running at priority 0 and raise the priority interrupt level to 1. Any further level 1 interrupts will not cause a preemption, but higher priority levels, say 10, will.

The interrupt priority level mechanism raises an integrity problem where a high priority device is given to a low security level application and a low priority device to a high security level application. The low security level application can starve the high application. Hence low security processes should not have access to devices with an interrupt priority that is higher than the device with the lowest interrupt priority held by a high security level application. In the Paramecium philosophy it is not up to the kernel to enforce this policy. Instead a separate service, a policy manager, should interpose the device manager interface and enforce the policy it sees fit.

An extra complication for interrupt handlers is that on a SPARC interrupts are level triggered rather than edge triggered. That is, an interrupt takes place when the interrupt line is high. The interrupt line continues to be high, and thus interrupts the processor, until the driver has told the device to stop interrupting. Hence to prevent an infinite amount of nested interrupts, the processor has to raise its interrupt priority level to that of the interrupt. This allows higher level interrupts to preempt the processor, but will mask out lower level interrupts.

Before an interrupt is turned into an event, the processor's interrupt priority level is raised to that of the interrupt. This is recorded in the event return information structures. Paramecium assumes that one of the first actions the driver will take is to acknowledge the interrupt to the device. It will then continue processing and eventually lower the priority interrupt level of the processor when:

1) The interrupt event returns normally.

2) The interrupt event performs a detach (swap) stack operation.

The first case is simple, the priority interrupt level is just restored. Most interrupts are handled like this. In the second case the priority interrupt level is also restored because the handler is returned to the event's inactive handler list for future interrupts. When all handlers for the event are busy, the interrupt level masks out any further interrupts of that level until one of the handlers returned or detached. The second case only occurs when using the thread package to turn device interrupts into real pop-up threads.

A side effect of preemptive events is that the kernel operations need to be nonblocking or the kernel has to implement a very fine grained locking mechanism to improve concurrency. We explored the later which, due to an architectural problem, caused a fair amount of overhead (see below). This overhead could probably be avoided by using *optimistic locking* techniques [Stodolsky *et al.*, 1993]. On the other hand, our kernel operations are sufficiently small and short enough that they might be atomic, such as in the OSKit [Ford *et al.*, 1997], and not require any locking. This has not been explored further in the current system.

Interfaces

The event and chain abstractions are exported by the kernel using two different interfaces. The event interface, see Figure 3.13, manages event creation, deletion, handler registration, invocation, and branching.

Method	**Description**
event_id = create(event_name)	Create new event
destroy(event_id)	Destroy event and release all handlers
enable(event_id)	Enable events
disable(event_id)	Disable events
handler_id = register(event_name, context_id, method, stack)	Register a new event handler
unregister(handler_id)	Unregister an event handler
result = invoke(event_id, arguments)	Explicitly invoke an event
branch(event_id, arguments)	Branch to an event
current_stack = detach(new_stack)	Detach current stack

Figure 3.13. Event interface.

The create method from the event interface is used to create an event with a public name evname. A private resource identifier is returned when the event was created successfully. The create method is used to define new events that can be used, for example, for interprocess communication or for MMU fault redirection. An event is deleted by calling destroy with its private resource identifier as an argument. Event invocation can be temporarily disabled and enabled by using enable and disable to prevent race conditions when manipulating sensitive data structures. Event handlers are registered using register and removed using unregister. Registering a handler requires as arguments an event name, an execution context, a method address, and a stack. It returns a handler identifier which should be used as an argument to unregister. An explicit event invocation is achieved by calling invoke. It takes a private event identifier, and an argument vector. The arguments are pushed onto the handler stack or passed in registers as dictated by the calling conventions. Invoke returns when the method handler returns.

The branch method is used in cases where control is not passed back to the invoker but to the previous invoker (see Figure 3.13). The resource held by the current handler are relinquished before the actual invocation is made.

Invoking an event causes the next inactive handler to be made active. The current handler can detach the stack, replace it with an unused one and deactivate the current handler using detach. This technique is used to turn events into pop-up threads.

The chain interface, see Figure 3.14, manages chain creation, deletion, and the swapping of chains. Its create method is used to create a new chain. The create method arguments are similar to the event invoke method. The create method returns the chain identifier for the new chain. The chain identifier for the current chain can be obtained by calling self. Chains are a thread of control abstraction that can cross multiple contexts and still behave as a single entity. A coroutine like interface exists to suspend the current chain and resume a new chain by calling the swap method. Finally, destroy is used to delete a chain. Destroying a chain causes all its resources to be relinquished.

Method	**Description**
chain_id = create(context_id, pc, stack, arguments)	Create a new chain
destroy(chain_id)	Destroy current chain
chain_id = self()	Obtain current chain
swap(next_chain_id)	Swap current by next chain

Figure 3.14. Execution chain interface.

Efficient Implementation on a SPARC

Our system has been implemented on a Sun SPARCClassic, a MicroSPARC with register windows. Register windows raise an interesting design challenge since they hold the top of the execution stack. Most operating systems, such as SunOS, Solaris, Amoeba, flush the register windows to memory before making the transition to a different protection domain. This causes a high overhead on IPC and interrupt processing.

In Paramecium we use a different mechanism where we mark windows as invalid, track their owner contexts, and flush them during normal window overflow and underflow handling rather than flushing them at transition time.

Independent from our event scheme, David Probert has developed a similar event mechanism for SPACE [Probert *et al.*, 1991]. In his thesis he describes an efficient cross-domain mechanism for the SPARC [Probert, 1996]. His implementation does not use register windows and thereby eliminates a lot of the complexity of the IPC code†. Unfortunately, by not using register windows you lose all the benefits of leaf optimizations. In our experience this leads to a considerable performance degradation. The Orca group reported a performance degradation which in some cases ran up to 15% for their run time system and applications [Langendoen, 1997].

†David Probert does mention in his thesis that he did work on a version for register windows but abandoned that work after a failed attempt.

The SPARC processor consists of an integer unit (IU) and a floating point unit (FPU) [Sun Microsystems Inc., 1992]. The IU contains, for the V8 SPARC, 120 general purpose 32-bit registers. Eight of these are global and the remaining 112 are organized into 7 sets of 16 registers. These 16 registers are further partitioned into 8 input and 8 local registers.

The eight global registers are available at any time during the execution of an instruction. Of the other registers a window of 24 registers is visible. Which window is visible depends on the *current window pointer* which is kept by the IU. The register window is partitioned into 8 input, 8 local, and 8 output registers. The 8 output registers overlap with the input registers from the adjacent window (see Figure 3.15). Hence, operating systems, like SunOS and Solaris, have to flush between 96 and 480 bytes per protection domain transition.

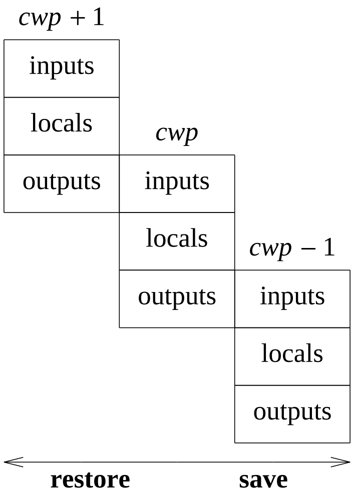


Figure 3.15. Overlapping register windows.

The SPARC IU maintains a current window pointer (*cwp*) and a *window invalid mask*. The current register window pointer is moved forward by the **restore** instruction and backward by the **save** instruction. Each function prologue executes a save instruction to advance the call frame on the stack; each epilogue executes a restore instruction to get back to the previous call frame. The window invalid mask contains a bit mask specifying which windows are valid and invalid. When a restore instruction advances into an invalid window a window underflow trap is generated. Similarly, a save instruction generates an overflow trap. These traps will then restore the next or save the previous register window and adjust the window invalid mask appropriately. At least one window is invalid and acts as a sentinel to signal the end of the circular buffer.

The general idea behind the efficient register window handling is to keep track of the ownership of a register set. The ownership is kept in an array that parallels the on-chip register windows. The array contains the hardware context number of the MMU context to which a register window belongs. On context switches, we denote the ownership change and mark the window of the old context as invalid and proceed. Marking it as invalid prevents other, possibly a user context, from accessing its contents.

The scenario above is slightly more complicated due to the fact that register windows overlap. That is, the 8 output registers in a window are the 8 input registers in the next window. They can therefore be modified. Rather than skipping two register windows, the 8 input registers are saved for trap events and restored when they return. There is no need to save them for other events because of the SPARC procedure call conventions. Violating these conventions only impacts the callee not the caller.

Unlike most operating systems, which flush all register windows, none of the register windows are saved to memory on a context switch. This is delayed to the normal window overflow handling which is performed as part of a normal procedure call. For very fast interrupt handling, *e.g.*, active message handling or simple remote operations, the interrupt code should restricts itself to one register window and thus prevent any saving to memory. The interrupt dispatching code ensures that at least one window is available; this is the window the interrupt handler is using.

On a return from a context switch, the reverse takes place. The invalid bit for the window where the context switch took place is cleared in the window invalid mask. The new MMU context is set to the value taken from its ownership record. During window overflow and underflow handling the ownership array and the window invalid mask are appropriately updated, especially where it involves invalid windows which are caused by context switches.

More formally, we keep the following invariants during context switch, window overflow, and underflow handling†:

$$\text{there is only one invalid window}_i \text{ for which owner}_i = -1 \tag{1}$$

This statement defines the empty window condition. At all times a single window remains empty because of the overlapping register window sets. This window is marked invalid and has the owner -1 which is especially reserved for the empty window slot.

$$\text{if window}_i \text{ is invalid then owner}_i \text{ is defined} \tag{2}$$

This statement defines so called transition slots. These are window slots which are marked invalid and are not the empty window slot. For these slots the ownership is defined. A transition slot denotes a transition between two MMU contexts, the next window beyond the transition belongs to the new context.

$$\text{if window}_i \text{ is invalid then} \begin{cases} \text{owner}_{i+1} \text{ is defined} \\ \text{owner}_{i-1} \text{ is defined} \end{cases} \tag{3}$$

†Without loss of generality we omit the *mod NWINDOWS* for the register window indices.

This condition defines that the ownership of the window slot surrounding an invalid window are defined. The window underflow, overflow, and cross protection domain transition code carefully preserves these invariants.

Describing the exact details -including all the pathological cases when traps end up in invalid windows- of the protection domain transition, underflow and overflow handling is beyond the scope of this thesis. However, in order to illustrate its complexity, the steps necessary to perform a window overflow trap, *i.e.*, trying to save a frame when the window is full, are show in Figure 3.16.

```
window_overflow:
      compute new window invalid mask
      save                            // get into next window
      if (%sp unaligned)              // catch misalignments
            handle error
      set MMU to ignore faults        // do not trap on faults
      set MMU context to owner[cwp]   // owner of this frame
      if (owner[cwp])                 // user stack frame
            verify stack lies in user space
      save registers to memory
      set MMU context to current      // back to current context
      if (MMU fault) handle error     // did a fault occur?
      set MMU to raise faults         // allow faults again
      restore                         // get into original window
      clear registers                 // clear any residue
      rtt                             // return from trap
```

Figure 3.16. Window overflow trap handling pseudo code.

Conceptually the handling of a window overflow trap is straightforward: 1) compute the new window invalid mask, 2) get into the window that needs to be saved, 3) save 16 registers into memory that belongs to the owning context, 4) get back to the previous window, 5) return from trap. Unfortunately, a SPARC V8 CPU does not allow nested traps and will reset the processor on a double fault. We therefore have to inline all the code that guards against faults such as alignment and memory violations.

Despite the complications of register window handling our technique for register window handling works reasonably well. On our target platform, a 50 MHz MicroSPARC, the time it took to make a cross protection domain event invocation to the kernel, *i.e.*, a null system call, was 9.5 μsec as opposed to 37 μsec for a similar operation on Solaris. A detailed analysis of the IPC performance is presented in Chapter 6. The null system call performance could conceivably be improved, since the code got convoluted after the initial highly tuned implementation.

Register window race condition

Inherent in the SPARC register window architecture is a subtle race condition that is exposed by our event driven architecture. Ordinarily, when an interrupt occurs the current thread of control is preempted, the register window is advanced into the next window, the preempted program counter and program status register are saved in the local registers in that new window and the program counter is set to the address of the low-level interrupt handler where execution continues. To return from an interrupt the program status register and the program counter are restored into the appropriate registers and a return from trap is issued after which the preempted thread continues execution. Now, consider the following code sequence to explicitly disable interrupts on a SPARC CPU:

```
mov     %psr, %l0             ! 1: get program status word into %l0
andn    %l0, PSR_ET, %l0      ! 2: turn off enable trap bit
mov     %l0, %psr             ! 3: set %l0 to program status word
```

This is the standard sequence of loading the program status register, masking out the enable trap bit, and setting it back. On a SPARC this requires three instructions because the program status register cannot be manipulated directly.

Since the program status register also contains the current window pointer an interesting race condition occurs. When an interrupt is granted between instructions 1 and 2 there is no guarantee in our event driven architecture that it will return in the same register window as before the interrupt. The reason for this is the event branch operation that short cuts event invocations without unwrapping the actual call chain. Note that all other state flags (e.g condition codes) and registers are restored on an interrupt return.

This race condition can be solved by using the property that interrupts are implicitly disabled when a processor trap occurs and setting the interrupt priority level. The interrupt priority level is stored in the program status register and controls which external sources can interrupt the processor. Groups of devices are assigned an interrupt priority level and when the current processor level is less than the device the interrupt is granted. The highest interrupt level is assigned to a nonmasking interrupt signaling a serious unrecoverable hardware problem. Consequently the highest priority can be used to effectively disable interrupts.

The race free implementation of disabling interrupts consists of a trap into the kernel (disabling the interrupts) followed by setting the high interrupt priority level (effectively disabling the interrupts) and return from the trap (enable interrupts again). Of course, the code for this implementation has to prevent that an arbitrary user program can disable interrupts at will. This is achieved by checking the source of the caller. The kernel is allowed to manipulate the interrupt status, any user process is not.

Other operating systems, like SunOS, Solaris, and Amoeba do not suffer from this race condition because of their two phase interrupt model (see Figure 3.17). The hard level interrupt handler deals with the interrupt in real time but does little more than querying the device and queuing a soft level interrupt. That is, no operations that

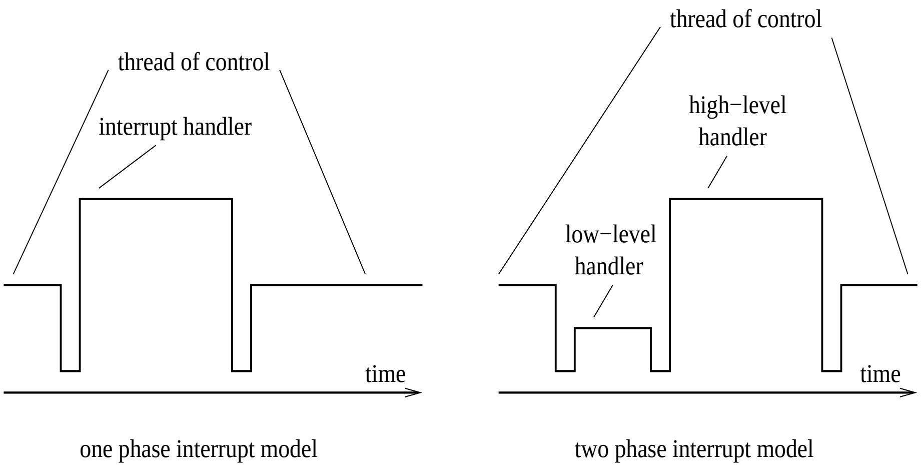


Figure 3.17. One *vs.* two phase interrupt models.

might change the current window pointer, like thread switches or long jumps, take place. The soft level interrupt handler gets called from the scheduler and can make current register window changes.

In the next section we discuss proxy interface invocations which allow object interfaces to be invoked from contexts that do not implement the actual object. The underlying technique for this is the event mechanism described above.

3.4.5. Naming and Object Invocations

Central to Paramecium is the name service from which interface pointers are obtained. Since this service is such an essential part of Paramecium it is implemented by the base kernel. The name server supports the name space mechanisms used by the object model, see Chapter 2, and a number of Paramecium specific additions. These additions provide support for multiple protection domains.

The object model name space is a single hierarchical space which stores references to all instantiated objects. Extending it to multiple protection domains raises the following issues:

- How to integrate the name space and multiple protection domains? Either each protection domain is given its own name space, which is disjoint from other domains, or there is a single shared name space where each protection domain has its own view on it.
- How are interfaces shared among multiple protection domains?

In Paramecium we decided to augment the name space mechanism by giving each protection domain a view of a single name space tree. This view consists of a subtree where the root is the start of the name space tree for the protection domain. Protection domains can traverse this subtree but never traverse up beyond their local root (see Figure 3.18). As is clear from this figure, the name space tree for a protection domain also contains as subtree the name space for the children it created.

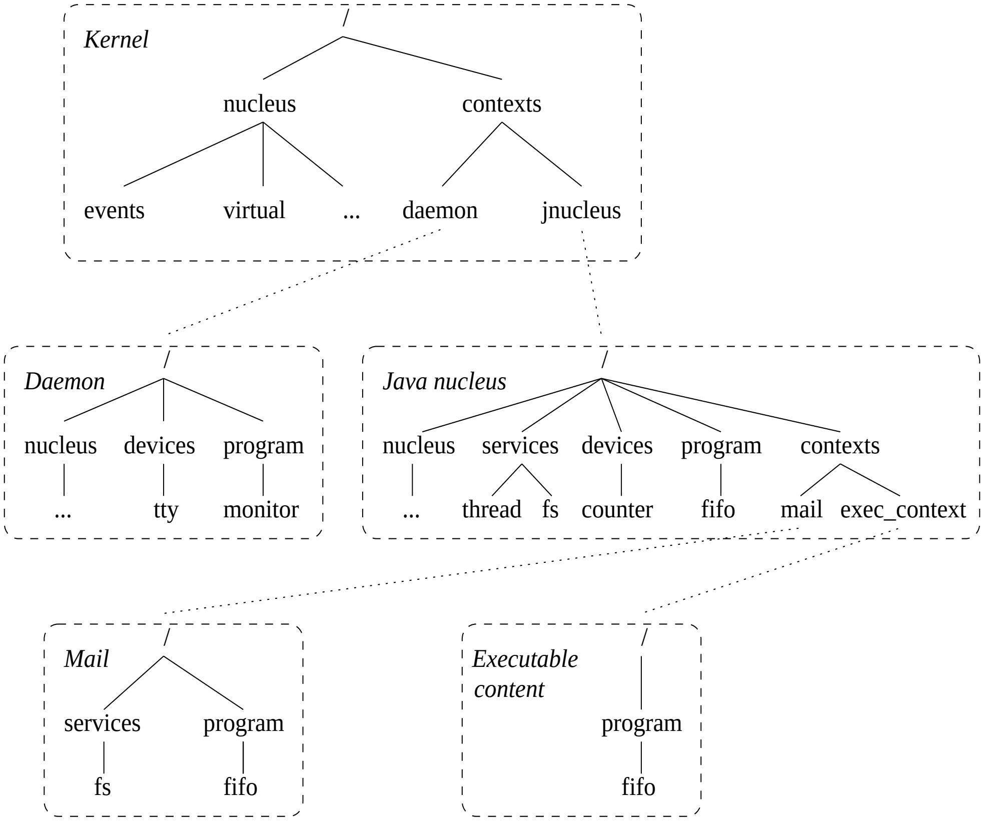


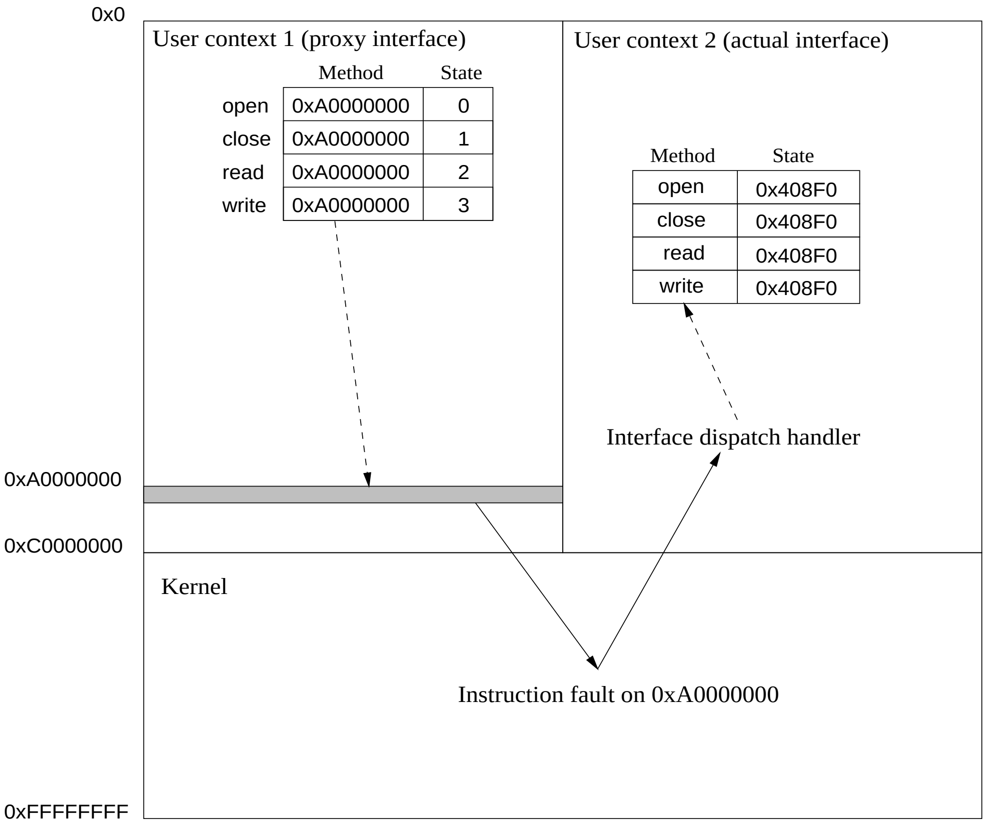
Figure 3.18. Paramecium name spaces. Each context has its own name space tree, here indicated by the dashed box. The contexts themselves form a tree with the kernel at the root.

Organizing the name space as a single tree rather than multiple disjoint trees make the management easier and more intuitive. For example, the initial name space for a protection domain is empty. It is created and populated by its parent, which designates one of its subdirectories as the root for the new protection domain. The parent can link to interfaces in its own subtree or install new interfaces that refer to its own objects. Since the kernel has full view of the name space, kernel components can access any interface in the system, including those that are private to the kernel.

When a request is made to bind to an interface that is not implemented by the requesting address space, *e.g.*, caused by a parent that linked one of its interfaces, the directory server will automatically instantiate a proxy interface. A proxy interface is an interface stub which turns its methods into IPCs to the actual interface methods in a different protection domain. Upon completion of the actual method, control is transfered back to the invoking context. This method is similar to surrogate objects in network objects [Birrell *et al.*, 1993], proxy objects by Marc Shapiro [Shapiro, 1986], and their implementation in COOL [Habert *et al.*, 1990].

A proxy interface consists of an ordinary interface table with the methods pointing to an empty virtual page and the state pointer holding the method index. Invoking the method causes a page fault on this empty page and transfers control to the interface dispatch routine in the receiving domain. This dispatch handler will use the index and invoke the appropriate method on the real object. The proxy interface is set up the first time a party binds to it.

For example, consider see Figure 3.19, where the write method is called on a proxy interface in context 1. Calling it will cause control to be transfered to address 0xA0000000 which is invalid. The handler for this fault event is the interface dispatcher in context 2. The dispatcher will lookup the actual interface using the fault address that is passed by the kernel as a parameter and invoke the write method on the actual interface and return control upon completion. The parameters are passed in registers and in previously agreed upon shared memory segments.


0x0
User context 1 (proxy interface)
User context 2 (actual interface)
Interface dispatch handler
0xA0000000
0xC0000000
Kernel
Instruction fault on 0xA0000000
0xFFFFFFFF

	Method	State
open	0xA0000000	0
close	0xA0000000	1
read	0xA0000000	2
write	0xA0000000	3

Method	State
open	0x408F0
close	0x408F0
read	0x408F0
write	0x408F0

Figure 3.19. A proxy interface method invocation.

In the current implementation, which is targeted at user/kernel interaction, no effort is made to swizzle [Wilson and Kakkad, 1992] pointer arguments. The kernel has full access to user level data anyway. For user to user interaction the current implementation assumes that the pointers to shared memory regions are setup prior to invoking the method. For a more transparent mechanism the IDL has to be extended using techniques like those used in Flick [Eide *et al.*, 1997] or use full fledged communication objects as in Globe [Van Steen *et al.*, 1999]. Neither have been explored in Paramecium.

The name service provides standard operations to bind to an existing object reference, to load an object from the repository, and to obtain an interface from a given object reference. Binding to an object happens at runtime. To reconfigure a particular service, you override its name. A search path mechanism exists to control groups of overrides. When an object is owned by a different address space the name service automatically instantiates proxy interfaces.

For example, consider the name space depicted in Figure 3.16. Here the jnucleus program created two subdirectories mail and exec_content. The first contains the mail application and the second is the executable content, say a Java [Gosling *et al.*, 1996] program. By convention the subtrees for different protection domains are created under the /contexts directory. The jnucleus domain populated the name space for mail with a services/fs interface that gives access to the file system and a program/fifo interface that is used to communicate with the executable content. The executable context domain, exec_content, only has access to the FIFO interface program/fifo to communicate with the mail context. It does not have access to the file system, or any other service. The mail context has access to the file system but not to, for example, the counter device devices/counter.

To continue this example, assume that the FIFO object is implemented by the jnucleus context. The name program/fifo in the contexts mail and exec_content is a link to the actual FIFO object in the jnucleus context. When the exec_content context binds to program/fifo by invoking bind on the name server interface (see Figure 3.20). The name server looks up the name and determines that it is implemented by a different protection domain, in this case jnucleus, and will create a proxy interface for it. How proxies are created is explained below.

The name server implements the interface in Figure 3.20. This interface can be divided into two parts: manipulations and traversal operations. The former manipulates the name space and the latter examines it. Most operations work on a current directory, called a name server context, which is maintained by the application and passed as a first parameter. To prevent inconsistencies, we made all the name space operations atomic and ensured that any interleaving of name space operations will still results in a consistent name space.

The bind operation searches the name space for a specified name starting in the current context using the standard search rules. These search rules are described in Chapter 2. To register an interface to an object, the register operation is used. The map operation loads an object from a file server and maps it into the context specified by the where parameter. When where is zero, it is mapped into the current context.

Method	**Description**
interface = bind(naming_context, name)	Bind to an existing name
unbind(naming_context, interface)	Unbind an interface
interface = map(naming_context, name, file, where)	Instantiate an object from a file
register(naming_context, name, interface)	Register an interface
delete(naming_context, name)	Delete a name
override(naming_context, to_name, from_name)	Add an override (alias)
new_naming_context = context(naming_context, name)	Create a new context
status(naming_context, status_buffer)	Obtain status information
naming_context = walk(naming_context, options)	Traverse the name space tree

Figure 3.20. Name service interface.

When an object is loaded into the kernel, it will check the object's digital signature as described in Section 3.3. For convenience, the current implementation uses a file name on an object store to locate an object representation. For this the kernel contains a small network file system client using the trivial file transfer protocol and UDP. A pure implementation should pass on a stream object rather than a file name. All three methods described above return a standard object interface to the object at hand or *nil* if none is found.

The unbind and delete operations remove interfaces from the name space either by interface pointer or name. The main difference between the two operations is that unbind decrements a reference count kept by bind and only deletes the name if the count reaches zero. Delete on the other hand forcefully removes the name without consulting the reference count. Introducing a link in the name space is achieved by the override operation, and the context operation creates a new context.

Examining the name space is achieved by the status and walk operations. The status operation returns information about an entry. This includes whether the entry is a directory, an interface, an override, or a local object. The walk operation is used to do a depth first walk of the name space starting at the specified entry and returning the next one. This allows all entries to be enumerated. The status and walk operations are the only way to examine the internal representation of the name space.

3.4.6. Device Manager

The Paramecium kernel does not contain any device driver implementations. Instead, drivers are implemented as modules which are instantiated on demand either in the user or the kernel address space. The advantage of user controlled device drivers is that they can be adapted to better suite the abstractions needed by the applications. In addition, running device drivers outside the kernel in their own context provides strong fault isolation. Unfortunately, the kernel still needs to be involved in the allocation of devices since they exhibit many inherent sharing properties. Hence, the kernel contains a device manager that controls the allocation of all the available devices and provides a rudimentary form of device locking to prevent concurrent access.

The interface of the device manager is modeled after the IEEE 1275 Open Boot Prom standard [IEEE, 1994]. Devices are organized in a hierarchical name space where each node contains the name of the device, its register locations, its interrupts, and a set of its properties. These properties include, device class, Ethernet address, SCSI identifier, *etc*. Examples of device names are counter for the timer device, le for the Lance Ethernet device, and esp for the SCSI device.

Step	**Action**	**Description**
1	Allocate DMA (*ledma*) device	Obtain an exclusive lock on this device
2	Get device registers	Used to communicate to the *ledma* device
3	Allocate Ethernet (*le*) device	Obtain an exclusive lock on this device
4	Get device registers	Used to communicate to the *le* device
5	Get interrupt event	Device interrupts generate this event
6	Allocate buffers	Transmit and receive buffers
7	Map buffers into I/O space	Allow *le* device to access the buffers

Figure 3.21. Steps involved in allocating an Ethernet device.

In Figure 3.21 we show the steps our Ethernet network device driver has to perform in order to access the actual network hardware. It first has to allocate the DMA ASIC (*ledma*), get its device registers, and then configure it appropriately. Then the device driver has to allocate the actual network device (*le*), get access to its device registers an obtain its interrupt event. The later is raised whenever a receive or transmit interrupt is generated by the hardware. The driver then proceeds by allocating the transmit and receive buffers which are then mapped into the I/O space which makes them available to the device hardware.

The steps above are all captured in the device interface which is obtained from the device manager (see Figure 3.22). This interface gives access to the device registers, using the register method, by mapping them into the requestor's address space and assists in setting up memory mapped I/O areas, using map and unmap methods. The device interrupts are accessed using the interrupt method. They return the event name for the interrupt. Additional properties, such as Ethernet address, SCSI identifier, or display dimensions are retrieved using the property method.

Method	**Description**
virtaddr = register(virthint, index)	Get device register address
event_id = interrupt(index)	Get device interrupt event
virtaddr = map(address, size)	Map memory into device I/O space
unmap(virtaddr, size)	Unmap memory from device I/O space
property(name, buffer)	Obtain additional device properties

Figure 3.22. Device interface.

Depending on a per device policy the device manager enforces an exclusive locking strategy or a shared locking strategy. Most devices are exclusively locked on a first come first served basis. That is, the first driver claiming the device will be granted access. Any further requests by drivers from different address spaces are denied until the device is released. The allocation of some devices implies the locking of others. For example, allocating the Lance Ethernet chip also locks the DMA ASIC controlling the Lance DMA channels.

Memory mapped I/O regions, the memory areas to which the device has direct access (DMA), are mapped and unmapped through the device interface. These regions contain device initialization blocks or DMA-able memory regions. The sun4m architecture supports a 4 GB address space but some devices, such as the Ethernet hardware, are only capable of handling 16 MB. Therefore, the sun4m architecture has a separate I/O MMU that maps the 32-bit host address space to the 24-bit device address space. The map method creates this mapping.

The sun4m I/O MMU is a straightforward translation table which maps I/O addresses to physical memory page addresses. Aside from this mapping between two spaces, it is also used for security. A device cannot access a page when it is not mapped into the I/O MMU translation table. A very simple and useful extension would be to add support for multiple contexts and possibly read/write protection bits. This would make it possible to share devices that use a single shared resource such as DMA. Currently, when a single device is allocated that uses DMA all devices that use DMA are locked because the DMA controller is a single shared resource. By using multiple translation tables, one for each protection domain, each driver could manage its own DMA space, *i.e.*, the memory area from which the device can issue DMA requests, without interference from others. Even finer grained protection could be obtained by using the protection bits.

A system that is akin to this is the secure communications processor designed in 1976 by Honeywell Information Systems Inc. for the US Air Force [Broadbridge and Mekota, 1976]. This device sat between the main processor and the I/O bus and enforced a Multics ring style protection scheme [Organick, 1972] for accessing devices. It was used in SCOMP [Bonneau, 1978], the first system to get an Orange Book A1 security rating.

3.4.7. Additional Services

Besides the five services described in the sections above, two minor services exist. These are implemented in the kernel because they rely on internal kernel state. These services are a primitive identification service and the random number generator.

The identification interface returns the name (*i.e.*, public resource identifier) of the context that caused the event handler to be invoked. It does this by examining the last entry on the invocation chain. Using the context name an application can implement authentication and enforce access control. More complicated forms of identification, like traversing the entire invocation chain to find all delegations, have not been explored.

Obtaining strong random numbers without random number hardware support is one of the hardest quests in computer science. In Paramecium strong random numbers are especially important because they are the protection mechanism against guessing or fabricating resource identifiers. Paramecium uses a cryptographically pseudo random number generator [Menezes *et al.*, 1997]. In this scheme the random numbers are taken from an ordinary congruental modulo generator after which the result is passed through a one way function. The result of this function is the final random number and cannot be used to determine the state of the generator.

The algorithm above reduces the problem to generating a good initial seed for the congruental modulo generator. This seed is taken from as many high entropy sources as possible, for which the kernel is in the best position to obtain them. The initial seed is based on the kernel’s secret key, high resolution timer, total number of interrupts since kernel instantiation, *etc*. To prevent long sequences of dependent random numbers the generator is periodically, currently after 1,000 random number requests, reinitialized with a new seed. Our generator passes the basic χ^2 random tests; more advanced tests have not been tried.

All the interfaces exported by the Paramecium kernel are listed in Figure 3.23. They comprise the four key abstractions supported by the kernel. These are address space and memory management, event management, name space management, and device management. An additional interface provides access to the secure random number generator.

Subsystem	**Exported interface(s)**
Address space and memory management	Context interface Physical memory interface Virtual memory interface
Event management	Event interface Execution chain interface Authentication interface
Name space management	Name service interface
Device management	Device interface
Miscellaneous	Random number interface

Figure 3.23. Overview of the kernel interfaces.

3.5. Embedded Systems

One of the most interesting areas to deploy extensible operating systems is that of embedded computer systems. Embedded computer systems typically run dedicated applications and usually operate under tight memory and processing cycles constraints. Examples of embedded computer systems are manufacturing line control systems, secure cryptographic coprocessors, network routers, and personal digital assistants (PDAs). All of these systems typically consist of a single-board computer with specialized I/O devices and its operating system and application software in (flash) ROM.

Embedded devices typically operate under tight memory and processing cycles constraints, because they are either manufactured under certain cost constraints or should work within certain environmental constraints. The later includes constraints on power consumption, battery life, and heat dispensation. Each of these dictate the amount of memory and computer cycles available on an embedded device. It is therefore desirable to fine tune the operating system, that is the kernel, and support services, to a bare minimum that is required by the embedded device and its applications. For example, most embedded applications do not need an extensive file system, virtual memory system, user authentication, or even protection domains.

To investigate the impact of embedded systems on the Paramecium kernel we have ported it to a Fujitsu SPARCLite [Fujitsu Microelectronics Inc., 1993] processor with 4 MB of memory. The processor is a SPARC V8 core lacking multiply/divide, floating point, and MMU support. The processor does distinguish between supervisor and user mode but this is useless for protection since there is no MMU support. That is, a user mode program can rewrite the interrupt table and force an interrupt to get into supervisor mode. At best it provides an extra hurdle to prevent run-away programs from creating havoc.

Since our embedded hardware does not support protection domains we rewrote the kernel to exclude all support for different hardware contexts. This resulted in a code reduction of 51%. Most of this reduction could be attributed to: 1) removal of context, virtual, and physical memory management; 2) removal of proxy interface generation; and 3) a bare bones register window handling mechanism. This dramatic reduction also underscores Paramecium's design principle that the kernel's most important function is protection.

Applications that do not require access to the removed interfaces ran without any modification. For those that did require them, such as the shell to create new processes, a dummy component was loaded first. This component created the missing interfaces and reported back that only one hardware context existed. By using this component to provide the missing interfaces the applications did not have to be modified to run on the embedded system.

At first glance it appears that Paramecium is very suited for embedded systems. There are, however, two problems: integrated into Paramecium is the concept of dynamic loading. Even though this could be useful for updates, most embedded systems lack connectivity or need to be self contained. A second problem is that Paramecium currently lacks any real-time guarantees [Burns and Wellings, 1990]. Soft real-time guarantees, such as earliest dead-line first (EDF), are easy to implement in the thread scheduler. Hard real-time guarantees are much harder to provide. In theory it should be possible since Paramecium's native kernel interfaces do not block and the application can tightly control any interrupt. Most real-time operating systems such as QNX [Hildebrand, 1992] and VxWorks [Wind River Systems Inc., 1999] provide only soft real-time guarantees.

3.6. Discussion and Comparison

The development of Paramecium was the result of our experience with the Amoeba distributed operating system kernel. With it, we try to push the limits of kernel minimalization and the dynamic composition of the system as whole. Our Amoeba experiences showed that, although a minimal kernel is clearly desirable, some application specific kernel extensions can dramatically improve the application's performance. For this reason we explored kernel extensibility. To make kernel extensibility as straightforward as possible we used a component as a unit of extensibility and used a trust relationship to express the confidence we have in the safety of the extension.

In retrospect, kernel extensions are hardly ever used in Paramecium, at least not for the applications we explored. Either the application and all its run-time components are instantiated as separate user processes or all reside in the kernel address space. Care has been taken to provide the same environment in the kernel as well as user address space such that components are unaware in which space they are instantiated. This was very beneficial for the embedded version of Paramecium, here all components essentially run in the kernel address space. A similar experience has been reported for application specific handlers (ASHes) in ExOS [Kaashoek, 1997].

Paramecium's event mechanism is similar to Probert's scheme in SPACE [Probert *et al.*, 1991] and Pebble [Gabber *et al.*, 1999]. The performance of a user-kernel event invocation, about 9.5 μsec, is 4 times faster than a Solaris system call on the same hardware. This is relatively slow compared to contemporary systems such as ExOS and L4/LavaOS. The main reason for this are the hardware peculiarities of the SPARC processor. On the other hand, Pebble showed that very reasonable results can be achieved on the Intel architecture. A major drawback of the preemptive event model is that all software should be aware that it can be preempted and should properly synchronize shared data accesses. Most of these idiosyncrasies are hidden by the thread package but programming can be quite tricky in cases where this package is not used.

Paramecium shares many traits with other operating systems and in some areas it is fundamentally different. The following subsections give a per operating system comparison for the systems that share the same philosophy or techniques as Paramecium. In order we compare ExOS, SPIN, Scout, Flux OSKit, L4/LavaOS, Pebble, and SPACE.

ExOS/ExoKernel

Paramecium and ExOS [Engler *et al.*, 1994] have similar goals but are very different in their design philosophy. ExOS securely multiplexes the hardware to the application program. Hence the application binary interface (ABI) represents the underlying hardware interface rather than the more traditional system call interface. The operating system functionality itself is situated in libraries, called library operating systems, that are linked together with the applications.

The Exokernel approach should not be confused with the virtual machine system 360/370 architecture from IBM [Seawright and Mackinnon, 1979]. The 360/370 VM architecture provides an idealized hardware abstraction to the application rather than providing access to the actual hardware.

The advantage of the ExOS approach is that applications have very low latency access to the underlying hardware and a complete control over the operating system and its implementation since it is part of the application's address space. Applications can replace, modify, and use specialized library operating systems for specific application domains [Engler *et al.*, 1995]. For example, special purpose library operating systems exist for parallel programming, UNIX emulation, WWW servers, *etc*.

The ExOS design philosophy differs in two major ways from traditional operating systems and these are also the source of its problems. The main problem with ExOS is the sharing of resources. Resources that are used by a single application, for example a disk by a file server, are relatively straightforward to manage by that application. However, when the disk is shared by multiple noncooperating applications, there is a need for an arbiter. The most obvious place for the arbiter to reside is the kernel; less obvious is its task given that it should only securely demultiplex the underlying hardware.

An arbiter for a disk has to implement some sort of access control list on disk blocks or extents. For this the Exokernel uses UDFs (untrusted deterministic functions) which translate the file system metadata into a simple form the kernel understands. The kernel uses this function, $owns\!-\!udf_T$ [Kaashoek *et al.*, 1997], to arbitrate the access to disk blocks and enforce access control. A different method of arbitration is used for a network device. Here the Exokernel uses a packet filter, DPF [Engler and Kaashoek, 1996], to demultiplex incoming messages. If we were to add support to the Exokernel for a shared crypto device with multiple key contexts, we would have to add a new arbitration method for managing the different key contexts. It seems that each device needs its own very unique way of arbitration to support sharing. Even the simple example of only one file system using the disk and one TCP/IP stack using the network is deceptive. Both use DMA channels which is a shared resource.

Even if DMA requests can be securely multiplexed over multiple DMA channels there is still the open issue of the DMA address range. On most systems a DMA request can be started to and from any physical memory location, hence compromising the security of the system. Preventing this requires secure arbitration on each DMA request which has a serious performance impact.

A second, minor, problem with the Exokernel approach is that the applications are very machine dependent since much of the operating system and device driver knowledge is built in. This is easily resolved, however, by introducing dynamic loading the machine specific library operating systems at run time.

SPIN

SPIN [Bershad *et al.*, 1995b], is an operating system being developed at the University of Washington. It combines research in operating systems, languages, and compilers to achieve:

- *Flexibility*. Arbitrary applications may customize the kernel by writing and installing extensions for it. These extensions are dynamically linked into the kernel's address space. Potentially, each procedure call can be extended. Extensions are written in Modula3 [Nelson, 1991] a type-safe language.
- *Safety*. The dynamic linking process enforces the type-safe language properties and restricts extensions from invoking critical kernel interfaces. This isolates run-away extensions. To determine whether an extension was generated by a trusted compiler it uses a straightforward digital signature mechanism.
- *Performance*. Application-specific extensions can improve the performance because they have low latency access to system resources and services without having to cross protection boundaries.

Just like Paramecium, SPIN is an event based system, but unlike our system, procedure calls are event invocations too. SPIN extensions are extra event handlers that can be placed at any point where a function is called, hence they provide a very fine grained extension mechanism. Event invocations are handled by the event dispatcher.

This dispatcher enforces access control between components and also evaluates event guards. Event guards are referential transparent [Ghezzi and Jazayeri, 1987] and determine the order in which events should be invoked.

To improve the performance of the system they explored compiler optimizations and run time code generation techniques. Despite these optimizations the microbenchmark performance numbers for SPIN do suggest there is room for further improvement (on a 133 MHz Alpha AXP 3000/400, a cross address space call is 84 μsec and protected in-kernel calls are 0.14 μsec).

Paramecium and SPIN both share the same high-level goals. The actual implementation and design philosophy are radically different.

Scout

Scout [Montz *et al.*, 1994] is a communication-oriented operating system targeted at network appliances such as set-on-top boxes, network attached disks, special purpose servers (web and file servers), or personal digital assistants (PDAs). These network appliances have several unique characteristics that suggest re-thinking some of the operating system design issues. These characteristics are:

- *Communication-oriented*. The main purpose of a network appliance is handling I/O. Unlike traditional operating systems, which are centered around computation-centric abstractions such as processes and tasks, Scout is structured around communication-oriented abstractions.
- *Specialized/Diverse functionality*. Network appliances are centered around one particular function, such as recording and compressing video, which suggest the use of an application specific operating system.
- *Predictable performance with scarce resources*. Network appliances are typically consumer electronic devices and to keep the cost down it cannot overcommit resources to meet all the application requirements. The means that the operating system has do a good job at providing predictable performance under a heavy load.

The Scout operating system is centered around the concept of a path. A *path* is a communication abstraction akin to the mechanisms found in the *x*-kernel [Hutchinson *et al.*, 1989]. Another important aspect of Scout is that it is configurable; a Scout instance is generated from a set of building-block modules. Its framework is general enough to support many different kinds of network appliances. The third important aspect of Scout is that it includes resource allocation and scheduling mechanisms that offer predictable performance guarantees under heavy load.

Scout is similar in philosophy to Paramecium, but Paramecium tries to be more general-purpose. Unlike Scout, Paramecium is not primarily targeted at a particular type of application. Our system therefore does not contain any built-in abstractions, such as paths or allocation of scheduling mechanisms. Both systems are constructed out of modules but Paramecium is constructed dynamically rather than statically.

Furthermore, Paramecium uses a digital signature mechanism for extensions to preserve kernel safety. Scout does not provide user-kernel mode protection, everything is running in kernel mode, hence Scout does not attempt to provide any kernel safety guarantees.

Flux OSKit

The Flux OSKit [Ford *et al.*, 1997] is an operating builders toolkit developed at the University of Utah. It consists of a large number of operating system component libraries that are linked together to form a kernel. As its component model it uses a derivative of COM [Microsoft Corporation and Digital Equipment Corporation, 1995].

The OSKit provides a minimal POSIX emulation within the kernel to enable the migration of user-level applications. An example of this is a kernelized version of a freely available Java Virtual Machine, Kaffe [Transvirtual Technologies Inc., 1998]. Other applications are a native SR [Andrews and Olsson, 1993] and ML [Milner *et al.*, 1990] implementation and the Fluke [Ford *et al.*, 1996] kernel. The Fluke kernel is a new operating system that efficiently supports the *nested process model*, which provides, among others, strong hierarchical resource management. The OSKit has also been used to create a more secure version called Flask [Spencer *et al.*, 1999] which provides a flexible policy director for enforcing mandatory access policies.

The Flux OSKit and Paramecium share the ideas of a common model in which components are written. Together these components form a toolbox from which the kernel is constructed. Paramecium constructs the kernel dynamically while the OSKit uses static linking. Using dynamic linking is useful in situations where the environment changes rapidly, such as personal digital assistants that run an MPEG player at one moment and a game at the next while working under very tight resource constraints.

L4/LavaOS

LavaOS [Jaeger *et al.*, 1998] and its predecessor L4 [Liedtke *et al.*, 1997] are microkernels designed by Jochen Liedtke. The kernel provides fast IPC, threads, tasks, and rudimentary page manipulation. Their IPC is based on the *rendez-vous* concept and occurs between two threads. Arguments are either copied or mapped.

L4/LavaOS achieves remarkable fast IPC timings [Liedtke *et al.*, 1997]. For example, the current LavaOS kernel on an Intel Pentium Pro achieves a user to user domain transfer in about 125 cycles for small address spaces and about 350 cycles for large address spaces. The difference in performance is due to a clever segment register trick that prevents a TLB flush on an Intel Pentium Pro. This only works for small processes that are smaller than 64 KB.

The L4/LavaOS designers do not believe in colocating services into the kernel address space. Instead they keep the kernel functionality limited to a small number of fixed services. Other services, such as TCP/IP and virtual memory, run as separate processes on top of L4/LavaOS. To show the flexibility of the L4/LavaOS kernel researchers have modified Linux [Maxwell, 1999], a UNIX look alike, to run on top of it. Throughput benchmarks showed that a L^{4}Linux kernel achieves only a 5% degradation compared to native Linux [Härtig *et al.*, 1997]. This should be compared to a factor of 7 for MkLinux [Des Places *et al.*, 1996]. MkLinux is a similar attempt but uses the Mach microkernel instead. Currently, work is on its way to separate the monolithic Linux subsystem into a component based system. This is similar to the multiserver UNIX attempt for Mach.

The L4/LavaOS kernel is substantially different from Paramecium. It provides threads as its basic execution abstraction and synchronous IPC is based on *rendez-vous* between threads. Paramecium uses events and is asynchronous. L4/LavaOS kernel provides a *clans and chiefs* abstraction whereby among a group of processes one is assigned as chief. This chief will receive all IPCs for the group and forward it to the intended recipient [Liedtke, 1992]. This mechanism can be used to enforce access control, rate control, and load balancing. Paramecium does not have a similar mechanism. In Paramecium events can have a variable number of arguments these are passed in registers and spillage is stored on the handlers stack. In L4/LavaOS message buffers are transfered between sender and recipient. This buffer is either copied or mapped.

In Paramecium threads are implemented as a separate package on top of the existing kernel primitives. LavaOS provides them as one of its basic services.

Pebble

Pebble [Gabber *et al.*, 1999] is a new operating system currently being designed and implemented at Lucent Bell Laboratories by Eran Gabber and colleagues. Pebble's goals are similar to Paramecium: flexibility, safety, and performance. The Pebble architecture consists of a minimal kernel that provides IPC and context switches and replaceable user-level components that implement all system services.

Pebble's IPC mechanism is based on Probert's thesis work [Probert, 1996] which in turn is similar to Paramecium's IPC mechanism. Pebble does not have a concept for generic kernel extensions but it does use dynamic code generation, similar to Synthesis [Massalin, 1992], to optimize cross protection domain control transfers. These extensions are written in a rudimentary IDL description. Like the L4/LavaOS designers, the Pebble designers assume that their efficient IPC mechanism reduces the cost of making cross protection domain procedure calls, and therefore obviates the need for colocating servers in a single address space.

Since Pebble is still in its early development stage all the current work has been focussed on its kernel and improving its IPC performance (currently 110-130 cycles on a MIPS R5000). The replaceable user-level components are still unexplored.

SPACE

SPACE [Probert, 1996] is more of a kernel substrate than an operating system. It was developed by David Probert and John Bruno. SPACE focused on exploring a number of key abstractions, such as IPC, colocating protection domains in a single address space, threads, and blurring the distinction between kernel and user-level. It did not consider extensibility. As pointed out in this chapter, a number of these ideas were independently conceived and explored in Paramecium.

The SPACE kernel has been implemented on a 40 MHz SuperSPARC (superscalar) processor and does not use register windows. It achieves a cross protection domain call and return (*i.e.*, 2 portal traversals) of 5 μsec compared to 9 μsec for a Solaris null system call on the same hardware. Probert claims in his thesis [Probert, 1996] that not using register windows results in a slow down of 5% for an average application. In our experience this is more in the range of 15% [Langendoen, 1997] due to the missed leaf call optimizations. Paramecium does use register windows which adds a considerable amount of complexity to the event management code, but still achieves a cross domain call of 9.5 μsec on much slower hardware. As a comparison, Solaris achieves 37 μsec for a null system call on the same hardware.

SPACE its thread model is directly layered on top of its portal transition scheme. Since threads do not have state stored in register windows the context switches do not require kernel interaction. Hence, there are no kernel primitives for switching and resuming activation chains as in Paramecium.

Miscellaneous

Besides the operating systems mentioned above, Paramecium has its roots in a large number of different operating system projects. Configurable operating systems have long been a holy grail, and static configuration has been explored in object oriented operating system designs such as Choices [Campbell *et al.*, 1987], Spring [Mitchell *et al.*, 1994], and PEACE [Schröder-Preikschat, 1994]. Dynamic extensions have first been explored for specific modules, usually device drivers, in Sun’s Solaris and USL’s System V [Goodheart and Cox, 1994]. Oberon [Wirth and Gütknecht, 1992], a modular operating system without hardware protection, used modules and dynamic loading features of its corresponding programming language to extend its system.

Application specific operating systems include all kinds of special purpose operating systems usually designed by hardware developers to support their embedded devices such as set-on-top boxes, PDAs, and network routers. Examples of these operating systems are QNX [Hildebrand, 1992], PalmOS [Palm Inc., 2000], and VxWorks [Wind River Systems Inc., 1999].

Apertos [Lea *et al.*, 1995] was an operating system project at Sony to explore the application of meta objects in the operating system. Similar to ExOS, the central focus was to isolate the policy in meta objects and the mechanism in the objects. Recovering from extension failures in an operating system was the focus of Vino [Seltzer *et al.*, 1996], it even introduced the notion a transactions that could be rolled back in case of a disaster.

Events [Reiss, 1990; Sullivan and Notkin, 1992] have a long history for efficiently implementing systems where the relation between components is established at run time. They are used in operating systems [Bershad *et al.*, 1995b; Bhatti and Schlichting, 1995], windowing systems, and database systems. Paramecium's event mechanism is similar.

Paramecium uses sparse capabilities as resource identifiers. Capabilities have been researched and used in many systems. An extensive overview of early capability systems is given in [Levy, 1984]. Examples of such systems are Plessey System 250 [England, 1975], CAP [Wilkes and Needham, 1979], Hydra [Wulf and Harbison, 1981], KeyKOS [Hardy, 1995], AS/400 [Soltis, 1997], Amoeba [Tanenbaum *et al.*, 1986], and more recently Eros [Shapiro *et al.*, 1999].

Notes

Part of this chapter was published in the proceedings of the fifth Hot Topics in Operating Systems (HotOS) Workshop, in 1995 [Van Doorn *et al.*, 1995].

4

Operating System Extensions

This chapter describes a number of tool box components, such as a thread implementation, a TCP/IP implementation, and an active filter implementation. Traditionally these services are part of the operating system kernel, but in Paramecium they are separate components which are loaded dynamically on demand by applications that use them. The advantage of implementing them as separate dynamic components is that they are only loaded when needed, individual components are easier to test and the individual components are amenable to adaptation when required.

The first example of a system extension component is our thread package. It provides a migrating thread implementation with additional support for pop-up threads. Pop-up threads are an alternative abstraction for interrupts and to implement them efficiently, *i.e.*, without creating a new thread for each interrupt, we have used techniques similar to *optimistic active messages* [Wallach *et al.*, 1995].

Our thread package can run either inside the kernel address space or as a component of an user-level application, effectively forming a kernel-level or user-level thread implementation. Kernel-level thread implementations typically require a user application to make system calls for every synchronization operation. This introduces a performance penalty. To overcome this penalty, we use a state sharing technique which enables user processes to perform the synchronization operations locally without calling the kernel while the thread package is still implemented inside the kernel.

A second example of a system extension component is our TCP/IP implementation. This TCP/IP network stack is a multithreaded implementation using the thread package and pop-up threads described above. Central to the network stack implementation is a fast buffer component that allows copy-less sharing of data buffers among different protection domains.

As a final example of system extensions we describe an active filter scheme where events trigger filters that may have side effects. This work finds its origin in some of our earlier ideas on intelligent I/O adapters and group communication using active messages [Van Doorn and Tanenbaum, 1994]. In this chapter we generalize that work by providing a generic event demultiplexing service using active filters.

The examples in this chapter show the versatility of our extensible nucleus. The main thesis contributions in this chapter are: an extensible thread package with efficient pop-up semantics for interrupt handling, a component based TCP/IP stack and an efficient data sharing mechanism across multiple protection domains, and an active filter mechanism as a generic demultiplexing service.

4.1. Unified Migrating Threads

Threads are an abstraction for dividing a computation into multiple concurrent processes or *threads of control* and they are a fundamental method of separating concerns. For example, management of a terminal by an operating system is naturally modeled as a producer, the thread reading the data from the terminal and performing all the terminal specific processing, and a consumer, the application thread reading and taking action on the input. Combining these two functions into one leads to a more complicated program because of the convolution of the two concerns, namely terminal processing and application input handling.

In the sections below we will give an overview of our thread system for Paramecium and discuss the two most important design goals. These are:

- The ability to provide a unified synchronous programming model for events and threads, as opposed to the asynchronous event model provided by the kernel.
- The integration of multiple closely cooperating protection domains within a single thread system.

The overview section is followed by a more detailed discussion on the mechanisms used to implement these design goals. These mechanisms are active messages and pop-up threads and synchronization state sharing among multiple protection domains.

4.1.1. Thread System Overview

Traditionally operating systems provide only a single thread of control per process, but more contemporary operating systems such as Amoeba [Tanenbaum *et al.*, 1991], Mach [Accetta *et al.*, 1986], Windows NT [Custer, 1993] and Solaris [Vahalla, 1996] provide multiple threads of control per process. This enables a synchronous programming model whereby each thread can use blocking primitives without blocking the entire process. Systems without threads have to resort to an asynchronous programming model to accomplish this. While such systems are arguably harder to program, they tend to be more efficient since they do not have the thread handling overhead.

Reducing the overhead induced by a thread system has been the topic of much research and even the source of a controversy between so called *kernel-level* and *user-level* thread systems in the late 80’s and early 90’s (see Figure 4.1 for an overview of the arguments). Kernel-level thread systems are implemented as part of the kernel and have the advantage that they can interact easily with the process scheduling mechanism. The disadvantage of kernel-level threads is that synchronization and thread scheduling from an application requires frequent kernel interaction and consequently adds a considerable overhead in the form of extra system calls. Examples of kernel-level thread systems are Mach [Accetta *et al.*, 1986], Amoeba, and Topaz [Thacker *et al.*, 1988]. A user-level thread system on the other hand implements the thread scheduler and synchronization primitives as part of the application run-time system and has a lower overhead for these operations. The disadvantage of user-level threads is their poor interaction with the kernel scheduler, their poor I/O integration, and contention for the kernel if it is not thread aware. Thread-unaware kernels form a single resource which allows only one thread per process to enter, others have to wait in the mean time. Examples of user-level thread systems are FastThreads [Anderson *et al.*, 1989] and PCR [Weiser *et al.*, 1989].

	Kernel-level threads	**User-level threads**
Performance	Moderate	Very good
Kernel interaction	Many system calls	None
Integration with process management	Easy	Hard
Blocks whole process on page fault	No	Yes
Blocks whole process on I/O	No	Yes

Figure 4.1. Kernel-level *vs.* user-level threads arguments.

Various researchers have addressed the short comings of both thread systems and have come up with various solutions [Anderson *et al.*, 1991; Bershad *et al.*, 1992]. One solution is to create a hybrid version where a user-level thread system is implemented on top of a kernel-level system. These systems, however, suffer from exactly the same performance and integration problems. A different solution is *scheduler activations* [Anderson *et al.*, 1991] whereby the kernel scheduler informs the user-level thread scheduler of kernel events, such as I/O completion, that take place. This approach provides good performance and integration with the rest of the system, but requires cooperation from the application's run-time system.

Unfortunately, user-level thread systems are built on the assumption that switching threads in user space can be done relatively efficiently. This is not generally true. The platform for which Paramecium was designed, the SPARC processor [Sun Microsystems Inc., 1992], uses register windows which need to be flushed on every thread switch. Besides the fact that this is a costly operation since it may involve many memory operations, it is also a privileged operation. That is, it can only be executed by the kernel. As shown in Figure 4.2, where we compare thread switching costs for a user-level thread package on different platforms (only the SPARC processor has register windows) [Keppel, 1993], these hardware constraints can have a serious impact on the choice of thread system: user or kernel level. Namely, if you already have to call the kernel to switch threads why not perform other functions while you are there?

Platform	**Integer switch** (in µsec)	**Integer+floating point switch** (in µsec)
AXP	1.0	2.0
i386	10.4	10.4
MIPS R3000	6.2	14.6
SPARC 4-65	32.3	32.7

Figure 4.2. Comparison of thread switching cost (integer registers, and integer plus floating point registers) for some well-known architectures [Keppel, 1993].

These architectural constraints dictate, to some degree, the design and the use of Paramecium's thread package. More specifically, the actual design goals of Paramecium's thread package were:

- To provide an efficient unified synchronous programming environment for events and multiple threads as opposed to the asynchronous event model provided by the kernel.
- To provide an integrated model for closely cooperating protection domains using a single thread system.

The key goal of our thread package is to provide a simpler to use synchronous execution environment over the harder to use, but more efficient, asynchronous mechanisms offered by the kernel. The thread system does this by efficiently promoting events and interrupts to pop-up threads after which they behave like full threads. The pop-up mechanism borrows heavily from optimistic active messages and lazy task creation techniques which are described further below.

An important aspect of the thread package is enable a *lightweight protection model* where an application is divided into multiple lightweight protection domains that cooperate closely. This enables, for example, a web server to isolate the Java servlets (*i.e.*, little Java programs that execute on behalf of the client on the server) from the server proper and other servlets using strong hardware separation. To support this model, we need to provide a seamless transfer of control between cooperating protection domains and an efficient sharing mechanism for shared memory and synchronization state. The former is provided by using migrating threads, a technique to logically continue the thread of control into another protection domain, and the later is provided by sharing the state as well as some of the internals of the synchronization state. The resources are controlled by the server which acts as the resource manager.

The advantage of migrating threads is that they reduce the *rendez-vous* overhead traditionally found in process based systems. Rather than unlocking a mutex or signaling a condition variable to wakeup the thread in another process, the thread logically continues, *i.e.*, migrates, into the other address space. Thread migration is further discussed below.

In line with Paramecium's component philosophy and unlike most thread systems, Paramecium's thread package is a separate module that can be instantiated either in user space or kernel space. By providing the thread package as a separate module rather than an integrated part of the kernel it is amenable to application specific adaptations and experimentation. Unfortunately, due to a SPARC architectural limitation a full user-level thread package is not possible so it uses the kernel chain mechanism (see Section 3.4.4) to swap between different threads.

A key concern in multithreaded programs is to synchronize the access to shared variables. Failure to properly synchronize access shared variables can lead to *race conditions*. A race condition is an anomalous behavior due to an unexpected dependence on the relative timing of events, in this case thread scheduling. To prevent race conditions most thread packages provide one or more synchronization primitives. A brief overview of the primitives provided by our thread package is given in Figure 4.3. They range from straightforward mutex operations to condition variables.

In the next sections we discuss active messages which is the technique used to implement pop-up threads. This discussion is followed by a section on thread migration and a section on synchronization state sharing. These are the key mechanisms used by our thread package.

4.1.2. Active Messages

Active messages [Von Eicken *et al.*, 1992] are a technique to integrate communications and computation. They provide very low-latency communication by calling the message handler immediately upon receipt of the message. The address of the handler is carried in the message header and the handler is called with the message body as argument. The key difference between traditional message passing protocols is that with active messages the handler in message header is called directly when the message arrives. Usually, directly from the interrupt handler. Since the active messages contains the handler address, it requires the receiver to trust the sender not to supply it with an incorrect address.

Active messages provide low-latency communication at the cost of sacrificing security and severely restricting the generality of the message handler. Since an active message carries the address of its handler, conceivably any code could be executed on receipt of the message, including code that modifies the security state of the receiving

Primitive	Description
Mutexes	A mutex is a mechanism to provide synchronized access to shared state. Threads lock the mutex, and when it succeeds they can access the shared state. Only one thread at the time can access the state. Other threads either wait for the lock to become free by polling the lock state or block after which they are awakened as soon as the lock becomes available.
Reader/writer mutexes	A reader/writer mutex is similar to an ordinary mutex but classifies shared state access into read or write access. A reader/writer mutex allows many readers but only one writer at the time. A writer is blocked until there are no more readers. This mechanism allows more concurrency than an ordinary mutex operation.
Semaphores	Semaphores allow up to a specified number of threads to access the data simultaneously where each thread will decrement the semaphore counter. When the counter reaches zero, *i.e.*, the limit is reached, the entering threads will block until one of the threads releases the semaphore by incrementing the count. Semaphores are useful for implementing shared buffers where one semaphore represents the amount of data consumed and the other amount of data produced. Mutexes are sometimes referred to as binary semaphores.
Condition variables	Condition variables are a mechanism to grab a mutex, test for a condition and, when the test fails, release the mutex and wait for the condition to change. This mechanism is especially useful for implementing monitors.
Barriers	Barriers are a mechanism for a number of predetermined threads to meet at a specific point in their processing. When a thread meets the barrier it is blocked until all threads reach the barrier. They then all continue. This is a useful mechanism to implement synchronization points after, for example, initialization.

Figure 4.3. Overview of thread synchronization primitives.

system. These security problems are easily remedied by introducing an extra level of indirection as in our work on active message for Amoeba [Van Doorn and Tanenbaum, 1994]. Here we replaced the handler address by an index into a table which contained the actual handler address. The table was set up before hand by the recipient of the active message. Unfortunately, the lack of generality proved to be a serious problem.

The active message handler is directly executed on receipt of the message and is typically invoked from an interrupt handler or by a routine that polls the network. This raises synchronization problems since the handler preempts any operation already executing on the receiving processor. This can lead to classical race conditions where the result of two concurrent processes incrementing a shared variable can be either 0, 1, or 2 depending on their execution schedule [Andrews and Olsson, 1993]. Hence, there is a need to synchronize shared data structures. However, active message handlers are not schedulable entities and can therefore not use traditional synchronization primitives. For example, consider the case where an active message handler grabs a mutex for which there is contention. It would have to block, but it cannot because there is no thread associated with the handler. Various solutions have been proposed for this problem. For example, in our active message work for Amoeba, we associated a single lock with an active message handler. When there was no contention for the lock the handler would be directly executed from the interrupt handler. In the event of contention a continuation would be left with the lock [Van Doorn and Tanenbaum, 1994]. This approach made the handler code less restrictive than the original active message design, but it was more restrictive than the elegant technique proposed by Wallach *et al.* called *optimistic active messages* (OAM) [Wallach *et al.*, 1995].

The optimistic active message technique combines the efficiency of active messages with no restrictions on the expressiveness of the handler code. That is, the handler may use an arbitrary number of synchronization primitives and use an arbitrary amount of time to process the handler. The technique is called *optimistic* in that it assumes that the handler will not block on an synchronization primitive. If it does, the handler will be turned into a full thread and rescheduled. Similarly, when the active message handler has used its time-slice it is promoted to a full thread as well. This technique can be thought of as a form of *lazy* thread creation [Mohr *et al.*, 1992].

4.1.3. Pop-up Thread Promotion

In addition to the traditional thread operations discussed in section 4.1.1, we added support to our thread system to handle pop-up threads. Pop-up threads are used to turn events into full threads but with the exception that the thread is created on-demand as in optimistic active messages [Wallach *et al.*, 1995] and lazy task creation. The pop-up thread mechanism can be used to turn any event into a full thread, not just interrupt events, and are used to hide the asynchronous behavior of Paramecium's event scheme.

To illustrate this, Figure 4.4 shows a typical time line for the creation of a pop-up thread. In this time line a thread raises an event and continues execution in a different protection domain. At some point the event handler is promoted to a pop-up thread. This causes a second thread to be created and the original thread will return from raising the event.

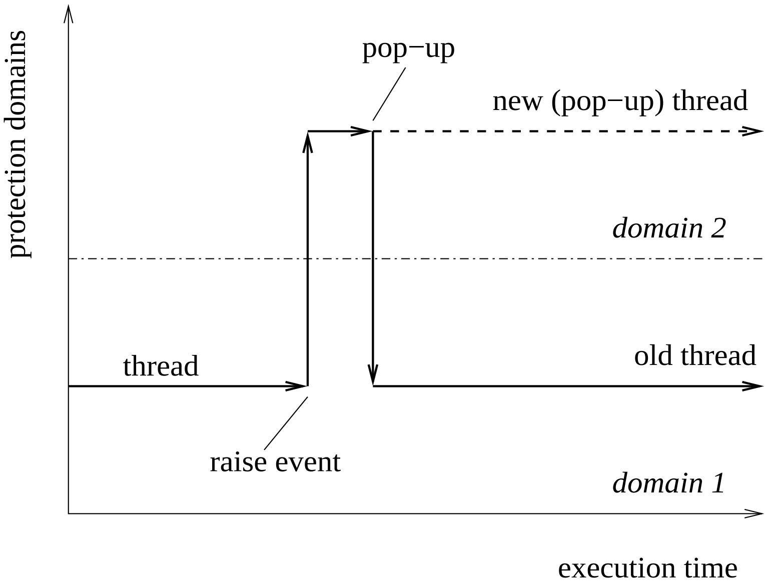


Figure 4.4. Pop-up thread creation time line.

To further illustrate this, Figure 4.5 contains a code fragment of an event handler from a test program. This handler is executed when its event is raised and it will perform certain test functions depending on some global flags set by the main program. The main program is not shown in this figure. When the handler is invoked it will first register itself with the thread system, using popup method, and mark the current thread as pop-up (as with any event handlers are logically instantiated on top of the preempted thread of control, see Section 3.4.4). Like optimistic active messages, if the event handler blocks it will be promoted to a full thread and the scheduler is invoked to schedule a new thread to resume execution. If the handler doesn't block, it will continue execution and eventually clear the pop-up state and return from the event handler. Clearing the pop-up state is performed by the thread destroy method, destroy.

More precisely, an event handler is promoted to a full thread when it is marked as a pop-up thread and one of the following situation occurs:

1) The handler blocks on a synchronization variable. Each synchronization primitive checks whether it should promote a thread before suspending it.

2) The handler exceeded the allocated scheduler time-slice. In this case the scheduler will promote the handler to a full thread.

3) The handler explicitly promoted itself to a full thread. This is useful for device drivers handling interrupts. A driver first performs the low-level interrupt handling after which it promotes itself to a thread and shepherds the interrupt through, for example, a network protocol stack.

```
void
event_handler(void)
{
    thr−>popup("event thread");

    if (lock_test) // grab mutex for which there is contention
       mu−>lock();
    if (timeout_test) // wait for time-slice to pass then promote
       wait(timeout);
    if (promote_test) // promote event immediately to a full thread
       thr−>promote();

    thr−>destroy();
}
```

Figure 4.5. Example of an event handler using pop-up threads.

The execution flow of our thread system is shown Figure 4.6. Normal threads are started by the scheduler and return to the scheduler when they block on a synchronization variable or are preempted. When an event is raised the associated pop-up thread is logically executing on the preempted thread. If it doesn't block, or is preempted, it returns to the interrupted thread which continues normal execution. If it does block, or is preempted, control is passed to the scheduler which promotes the pop-up thread and disassociates it from the thread it was logically running on. These two threads are then scheduled separately as any other thread.

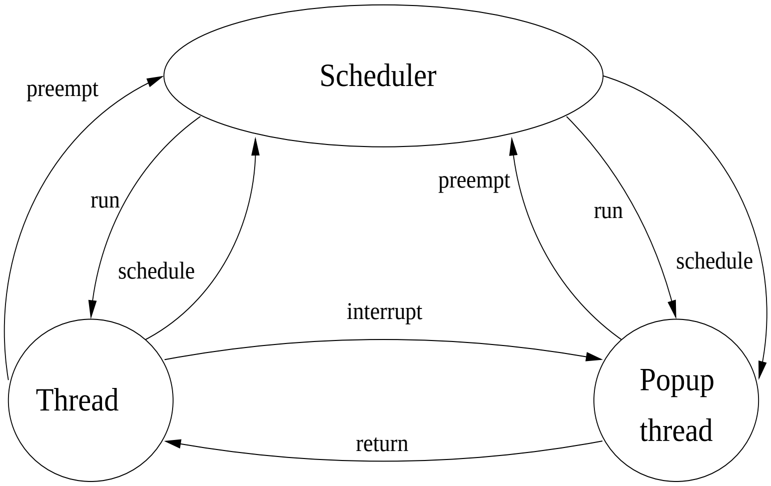


Figure 4.6. Thread execution control flow.

Promoting a pop-up thread to a full thread consists of allocating a new thread control block and detaching the event stack and replacing it with a new stack for the next event. The current event stack, which has all the automatic storage and procedure activation records on it, is used as the thread stack. This obviates the need to copy the stack and relocate the data structures and activation records.

The thread package uses a round-robin scheduling algorithm within a priority queue. The package support multiple priority queues. When a pop-up thread is promoted it is given the highest priority and the order in which they are registered (that is, when they announced themselves to the thread system using popup) is preserved. The latter is important for protocol stacks. For example, our TCP/IP implementation, which is described in the next section, uses pop-up threads to shepherd incoming packets through the TCP/IP stack. Failing to preserve the FIFO order in which the packets are delivered would lead to disastrous performance problems.

4.1.4. Thread Migration and Synchronization

Thread migration is an integral part of our thread system and is directly based on the underlying event chain mechanism. That is, when a thread raises an event, control is transfered to a different protection domain where it resumes execution. The logical thread of control is still the same and it is still under control of the scheduler. Raising an event does not create a new thread but rather continues the current thread. However, an event handler may fork a separate pop-up thread after which the old thread resumes as if the raised event finished and the pop-up thread continues the execution of the event handler (see previous section).

The advantage of having a single thread abstraction spanning multiple protection domains is that it ties the thread of control and Paramecium’s lightweight protection domain mechanisms closely together. A single application no longer consists of one process, but instead might span multiple processes in different protection domains to provide internal protection. Migrating threads are the tool to intertwine these different protection domains seamlessly. An extensive example of their use is the secure virtual Java machine as discussed in Chapter 5.

With migrating threads it is important to be able to efficiently implement the thread synchronization primitives, especially when the synchronization variables are shared among different protection domains to synchronize access to shared memory segments. These synchronization primitives are provided by the thread package and operate on state local to the thread package implementation. For example, the implementation of a mutual exclusion lock (see Figure 4.7) consists of the mutex lock state and a queue of threads currently blocked on the mutex. The lock implementation first tests the lock status of the mutex. If it is set, the lock is in effect, otherwise there is no contention for the lock. If the mutex is not locked, it is simply marked as locked and the operation proceeds. Setting the lock state is an atomic operation implemented, in this case, by an atomic exchange. When there is contention for the lock, that is the lock state is already set, the current thread is removed from the run queue and put on the waiting queue associated with the mutex. The scheduler is then called to activate a new thread. All these operations must be atomic, since multiple threads may grab the mutex concurrently. Hence, they are performed within a critical region.

```
LOCK(mutex):
    while (atomic_exchange(1, &mutex.locked)) {
        enter critical region
        if (mutex.locked) {
            remove current thread from the run queue
            put blocked thread on lock queue
            schedule a new thread
        }
        leave critical region
    }
```

Figure 4.7. Mutual exclusion lock implementation.

The mutual exclusion lock implementation shown in Figure 4.7 requires access to internal thread state information and is therefore most conveniently implemented as part of the thread package. Unfortunately, when the thread package is implemented in a different protection domain, say the kernel, each lock operation requires an expensive cross protection domain call. This has a big performance impact on the application using the thread system. In general, applications try to minimize the lock granularity to increase concurrency.

Some systems (notably Amoeba) that provide kernel-level thread implementations improve the efficiency of lock operations by providing a shim that first executes a *test-and-set* instruction on a local copy of the lock state and only when the lock is already set the shim invokes the real lock system operation. In effect they wrap the system call with an atomic exchange operation as is done with the thread queue management in Figure 4.7. Assuming that there is hardly any contention on a mutex, this reduces the number of calls to the actual lock implementation in the kernel. Of course, this wrapper technique is only useful in environments without migrating threads. With migrating threads the locks are shared among different protection domains and therefore state local to a single protection domain would cause inconsistencies and incorrect locking behavior.

To support a similar wrapper technique for a migrating threads package we need to share the lock state over multiple protection domains. Each lock operation would first perform a test-and-set operation on the shared lock state before invoking the actual lock operation in case there was contention for the mutex. Instead of introducing a new shared lock state it is more efficient to expose the thread package's internal lock state (see Figure 4.8) and operate on that before calling the actual lock call. To guard against mishaps, only the lock state is shared; the lock and run queues are still private to the thread package. In fact this is exactly what Paramecium's thread system does, only it does it transparently to the application.

Consider the case where the thread package is shared among different protection domains. The first instance of the thread package is instantiated in the kernel and its interfaces are registered in all cooperating contexts. This instance will act as a kernel-

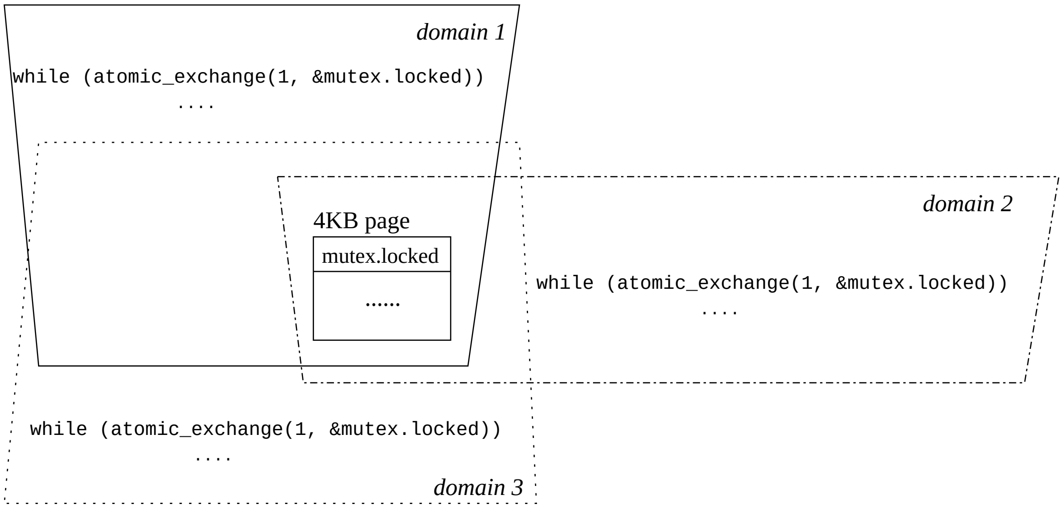

domain 1
while (atomic_exchange(1, &mutex.locked))
....
domain 2
4KB page
mutex.locked
......
while (atomic_exchange(1, &mutex.locked))
....
while (atomic_exchange(1, &mutex.locked))
....
domain 3

Figure 4.8. Synchronization state sharing among multiple protection domains.

level thread package, performing thread scheduling and the synchronization operations. Applications can use this thread package at the cost of additional overhead introduced by the cross domain invocations to the kernel. However, when a separate thread package is instantiated in the application context it will search, using the object name space, for an already existing thread package. If one exists, in this case the kernel version, it negotiates to share the lock state and performs the atomic test-and-set operations locally, exactly as described above. Only when the thread blocks will it call the kernel thread package instance.

The user-level and kernel-level thread package negotiations occur through a private interface. This interface contains methods that allow the user-level thread package to pass ownership for a specific region of its address space to the kernel-level thread package. The kernel-level thread package will use this region to map in the shared lock state. The region itself is obtained by calling the range method from the virtual memory interface (see Section 3.4.3) Once the lock state is shared, the user-level thread package uses an atomic exchange to determine whether a call to the kernel-level thread package is required.

In the current implementation the lock state is located on a separate memory page. This leads to memory fragmentation but provides the proper protection from accidentally overwriting memory. An alternate implementation, where multiple share lock states are stored on the same page, leads to false sharing and potential protection problems. We have not looked into hybrid solutions.

4.2. Network Protocols

Network connectivity is a central service of an operating system. It enables, for example, remote file transfer, remote login, e-mail, distributed objects, *etc*. Among the most popular network protocols is TCP/IP [Cerf and Kahn, 1974]. This protocol suite is used to connect many different kinds of machines operating systems and form the basis for the Internet. To enable network connectivity for Paramecium we have implemented our own TCP/IP stack.

The key focus of our TCP/IP stack is to take advantage of Paramecium's extensibility features, use the pop-up threads instead of interrupts (see Section 4.1.3), and provide efficient cross protection domain data transfer mechanisms. In the next two sections we discuss our copy-less cross protection domain data transfer mechanism and give an overview of our TCP/IP stack.

4.2.1. Cross Domain Shared Buffers

Traditional microkernel and server based operating systems suffer from two common performance bottlenecks. The first one is the IPC overhead, which becomes a concern when many IPCs are made to different servers. The second problem is the transport of data buffers between servers. In this section we focus on that problem. That is, how can we efficiently share data buffers among multiple cooperating protection domains. Our system is akin to Druschel's work on *Fbufs* [Druschel and Peterson, 1993] and Pai's work on IO-Lite [Pai *et al.*, 2000], with the difference that it is not hardwired into the kernel and allows mutable buffers. Our efficient transport mechanism is another example of Paramecium's support for the lightweight protection domain model.

The problem with multiserver systems, that is a system with multiple servers running on a single computer, is that each server keeps its own separate pool of buffers. These are well isolated from other servers, so that explicit data copies are required to transfer data from one protection domain to another. As Pai [Pai *et al.*, 2000] noted, this raises the following problems:

- *Redundant data copying*. Data may be copied a number of times when it traverses from one protection domain to another. Depending on the data size, this may incur a large overhead.
- *Multiple buffering*. The lack of integration causes data to be stored in multiple places. A typical example of this is a web page which is stored in the file system buffer cache and the network protocol buffers. Integrating the buffer systems could lead to storing only a single copy of the web page obviating the need for memory copies. This issue is less of a concern to us since we primarily use our buffer scheme to support our TCP/IP stack.
- *Lack of cross-subsystem optimization*. Separate buffering mechanisms make it difficult for individual servers to recognize opportunities for optimizations. For example, a network protocol stack could cache checksums over data

buffers if only it were able to efficiently recognize that a particular buffer was already checksummed.

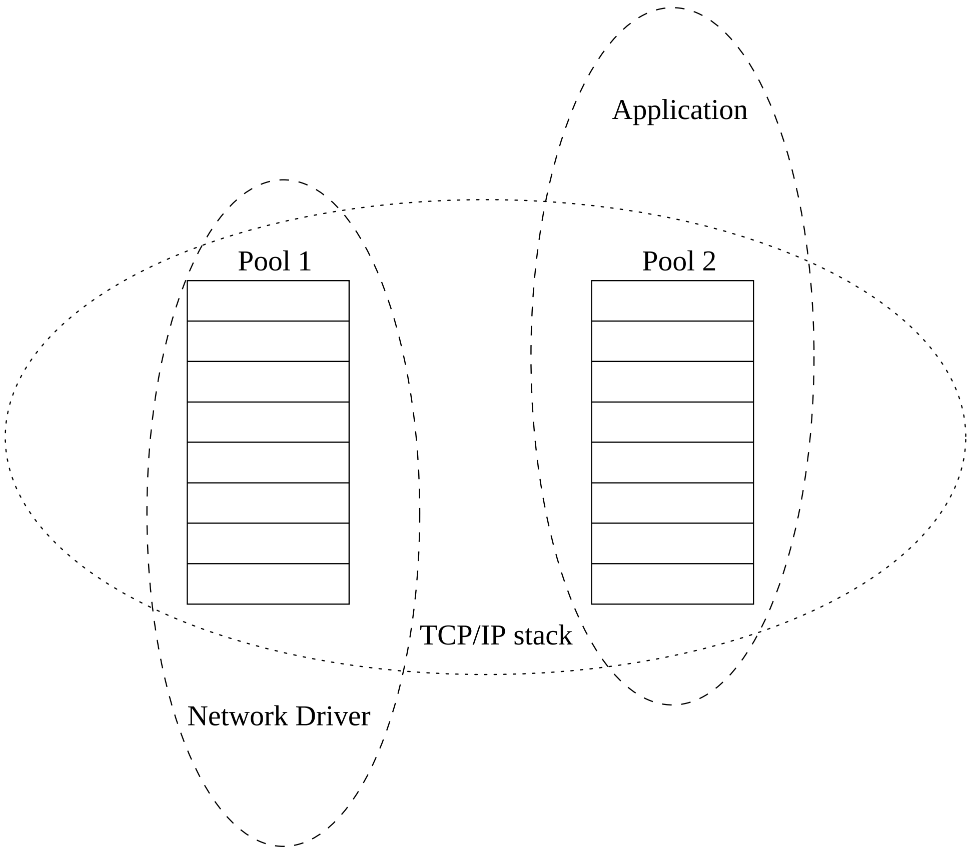


Figure 4.9. Cross domain shared buffers where buffer pool 1 is shared between the network driver and the TCP/IP module, and buffer pool 2 is shared between the TCP/IP module and the user application. Only the TCP/IP module has access to both pools.

To overcome these problems we have designed a shared buffer system whose primary goal is to provide an efficient copy-less data transfer mechanism among multiple cooperating protection domains. In our system, the shared buffers are mapped into each cooperating protection domain's virtual memory address space to allow efficient access. The shared buffers are mutable and, to amortize the cost of creating and mapping a shared buffer, the buffers are grouped into pools which form the sharing granularity. Every buffer pool has an access control list associated with it to control which domains have access to which buffers (see Figure 4.9). Our shared buffer mechanism is implemented as separate module and can be colocated with an application.

In order for a protection domain to use the shared buffer system it first has to register itself. By doing so, the protection domain relinquishes control over a small part of its 4 GB virtual memory address space, typically 16 MB, and passes it on to the shared buffer system. The buffer system will use this virtual memory range to map in the shared buffer pools. The buffer pools are only mapped into contexts that are authorized to have access to it. The buffer system guarantees that each shared pool is allocated on the same relative position in the virtual memory range of each participating protection domain. Hence, passing a shared buffer reference from one protection domain to another consists of passing an integer offset to the shared buffer in this virtual memory range instead of passing a pointer. To obtain a pointer to the shared buffer it suffices to add the base address of the virtual memory range which the buffer system uses to map in buffer pools.

To illustrate the use of the shared buffer system consider the following example where the network driver module allocates a shared buffer pool for incoming network packets and passes them on to the TCP/IP module. The interface to the shared buffer module is shown in Figure 4.10. In order for both the network driver module and the TCP/IP module to use the shared buffer system, they first have to register using the register method. This has as its argument a virtual memory range identifier that is obtained using the range method (see Section 3.4.3) and represents the part of the virtual address space that will be managed by the shared buffer system. The return value of the registration is the base address for all future shared buffers.

Method	**Description**
base_address = register(virtual_range_id)	Register with the buffer system
unregister()	Remove all associations
offset = create(nelem, elsize, elalign)	Create a buffer pool
destroy(offset)	Destroy a buffer pool
bind(offset)	Request access to a buffer pool
unbind(offset)	Release access from a buffer pool
add_access(offset, context_id)	Add context_id to the buffer pool access list
remove_access(offset, context_id)	Remove context_id from the access list
attribute(offset, flags)	Set buffer pool attributes

Figure 4.10. Shared buffer interface.

The next step is for the network driver to create a buffer pool for incoming messages,, using the create method. Usually, a small pool of buffers, say 128 buffers each 1514 bytes long, suffices for a standard Ethernet device driver. The buffer system will allocate a page-aligned buffer pool, map it into the virtual memory space of the device driver†, and return the offset to the buffer pool as a result. To access the actual buffer pool the network driver module has to add this offset to the base address it got when it registered. For the TCP/IP module to gain access to the shared buffer pool, the device driver module, which is the owner of the pool, has to add the module's context to the access control list using the add_access method. The TCP/IP module can then get access to the pool using the bind method. This method will, provided that the TCP/IP module is on the access control list, map the buffer pool into the TCP/IP module's address space. From then on passing buffers from this buffer pool between the network driver and the TCP/IP module only consists of passing offsets. No further binding or copying is required. For symmetry, each interface method described above has a complement method. These are unregister to remove access from the shared buffer system, destroy to destroy a shared buffer pool, unbind to release a shared buffer pool, and remove_access to remove a domain from the access control list. The attribute method associates certain attributes with a buffer pool. Currently only two attributes exist: The ability to map an entire pool into I/O space and to selectively disable caching for a buffer pool. Both attributes provide support for devices to directly access the buffer pools.

As identified above, the three main problems with nonunified buffer scheme's are: redundant data copying, multiple buffering, and lack of cross-subsystem optimization. Our system solves these problems by providing a single shared buffer scheme to which multiple protection domains have simultaneous access. Although the current example shows the network protocol stack making use of this scheme, it could also be used, for example, by a file system or a web server to reduce the amount of multiple buffering. Cross-subsystem optimizations, such as data cache optimizations, would be possible too. The example given by Pai [Pai *et al.*, 2000] to cache checksums is harder since the buffers in our scheme are mutable. Extending our buffer scheme with an option to mark buffers immutable is straightforward.

Our work is similar to Druschel's work on *Fbufs* and Pai's work on IO-Lite. In their work, however, the buffers are immutable and they use aggregates (a kind of scatter-gather list) to pass buffers from one domain to another. When passing an aggregate the kernel will map the buffers into the receiving address space, mark them read-only, and update the addresses in the aggregate list accordingly. In our scheme we use register and bind operations to gain access to the shared buffer pools instead of adding additional overhead to the cross domain IPC path. To amortize the cost of binding we share buffer pools rather than individual buffers as done in IO-Lite. This has a slight disadvantage that the protection guarantees in our system are of a coarser granularity than those in IO-Lite. Namely, we provide protection among buffer pools rather than individual buffers.

†The network device on our experimentation hardware can only access a limited portion, namely 16 MB, of the total 4 GB address space.

4.2.2. TCP/IP Protocol Stack

TCP/IP [Cerf and Kahn, 1974] is the popular name for a protocol stack that is used to connect different computers over unreliable communication networks. The name is derived from the two most important protocols among a suite of many different protocols. The IP [Postel, 1981a] protocol provides an unreliable datagram service, while TCP [Postel, 1981b] provides a reliable stream service on top of IP. The TCP/IP protocol suite forms the foundation of the Internet, a globe-spanning computer network, and is extensively described by Tanenbaum [Tanenbaum, 1988], Stevens [Stevens, 1994], Comer [Comer and Stevens, 1994], and many others. A typical use of a TCP/IP protocol stack is depicted in Figure 4.11. This example shows two hosts communicating over a network, Ethernet [Shock *et al.*, 1982], and using the TCP/IP stack to send a HTTP [Berners-Lee *et al.*, 1996] command from the browser to the WEB server.

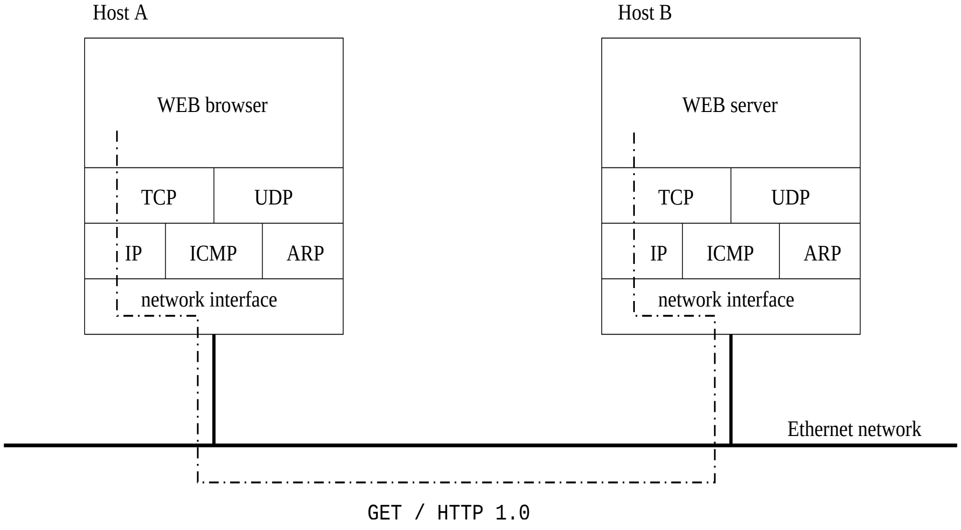


Figure 4.11. Example of a computer network with two TCP/IP hosts.

Rather than designing a TCP/IP protocol stack from scratch we based the Paramecium implementation on the Xinu TCP/IP stack [Comer and Stevens, 1994], with additions from the BSD NET2 stack [McKusick *et al.*, 1996], and heavily modified it. Our main modifications consisted of making the stack multithreaded, use pop-up threads to handle network interrupts, and use our shared buffer mechanism to pass data between modules. The TCP/IP protocol stack is implemented as a single module and depends on the availability of a network driver. Currently we have implemented an Ethernet driver module.

Since our protocol stack is implemented as a number of different components (network driver, TCP/IP stack, and shared buffers), various different configurations are possible. The configurations offer trade-offs between robustness and performance. For very robust systems strong isolation is an absolute necessity. Therefore, at the cost of extra IPC overhead, each component can be placed in its own separate protection domain and the amount of sharing can be minimized. That is, only a limited set of interfaces and buffer pools are shared. On the other hand, performance could improved by colocating the components in the application’s protection domain, thereby reducing the number of IPCs. The advantage of having the TCP/IP stack as a separate module is that it is more amenable to modification. One modification could be a tight integration of the TCP stack and a web server as in Kaashoek [Kaashoek *et al.*, 1997] where the web pages are preprocessed and laid out as TCP data streams that only require a checksum update before being transmitted. Another reason for user-level protocol processing is the performance improvement over a server based implementation [Maeda and Bershad, 1993].

The key modification to the Xinu TCP/IP stack was to turn it into a multithreaded stack. This mainly consisted of carefully synchronizing access to shared resources, such as the transmission control blocks holding the TCP state. The network driver uses the pop-up thread mechanism to handle network interrupts. These threads will follow the incoming message up the protocol stack until it is handed off to a different thread, usually the application thread reading from a TCP stream. Once the data is handed off, the thread is destroyed. This prevents the thread from having to traverse down the call chain where it will eventually still be destroyed by the network driver’s interrupt handler. This mechanism, the shepherding of incoming messages, is similar to the technique used by the X-kernel [Hutchinson *et al.*, 1989]. The addition of shared buffer support was straight forward since shared buffers are, after initialization, transparent with the existing local buffers.

4.3. Active Filters

In Chapter 3 we looked at extending the operating system kernel securely and used a mechanism based on digital signatures. In this section we look at a different and more restrictive way of extending a system: *active filters*. Active filters are an efficient event demultiplexing technique that uses application supplied predicate filters to determine the recipient of an event. The general idea is that an event producer, say a network device driver, uses the predicate filters to determine which process out of a group of server processes will receive the incoming network packet. This filter based demultiplexing mechanism can be used, for example, to balance the load among a group of web servers on the same machine. Whenever a web server is started, it first registers a predicate filter with the network device driver specifying under what load, number of requests currently being processed by that server, it is willing to accept new requests. The network device driver then uses this information to demultiplex the incoming network packets.

The example above points at one of main characteristics of active filters: the ability to share state with the user application that provided the filter, in this case the current load of the web server process. The other characteristic of active filters is that they are not confined to the local host but can of offloaded into intelligent I/O adapters which will do the demultiplexing and decide whether it is necessary to even interrupt the host. This requires filter specifications to be portable since the intelligent I/O adapters is likely to have a different processor than the host. Hence, our solution for extending the operating system by using code signing, as described in Chapter 3, does not work in this case since it does not provide the required portability, nor does it provide the flexibility and efficiency for relatively small and frequently changing filter expressions.

Active filters are an efficient event demultiplexing technique that uses simple predicate filters to determine the recipient of an event. The goal of this mechanism is to provide a generic event dispatching service that can be used throughout the system from demultiplexing requests to multiple servers for load balancing, to shared network event demultiplexing in the kernel, to implementing subject based addressing on intelligent I/O devices. The filters are called active because, unlike other systems, they may have side effects when they are evaluated.

Active filters find their origin in some of our early ideas on using intelligent network I/O devices to reduce the number of interrupts to the host processor by providing some additional filtering and limited processing capabilities. An example of this is a shared object implementation using a total ordered group communication protocol. Once a message has been delivered successfully to the host processor, retransmission attempts for that message can be safely ignored and host does not have to be interrupted again. In fact, for simple operations, like a simple shared integer object, the update could be handled entirely by the intelligent I/O device. Of course, this simple example leaves out many implementation details, but it does highlight the original ideas behind the filter scheme: user provided filter expressions and read/write access to message and user data.

A demultiplexing service that uses generic user filters with access to user and local memory raises the following issues:

- *Portable filter descriptions.* The filter specifications needs to be portable across multiple platforms and devices because a filter may be executing on a intelligent network device or on the host.
- *Security.* Since a filter expression is generic code which runs in another protection domain and is potentially hostile or buggy with respect to other filters, it needs to be strictly confined and controlled.
- *Efficient filter evaluation.* Since there may be many filters it is important to reduce event dispatching latency by having an efficient filter evaluation scheme.
- *Synchronizing local and user state accesses.* Since filters are allowed to access and modify local and user state, their accesses need to be synchronized with other concurrent threads in the system.

In our system we have addressed each of these issues as follows: For portability we use a small virtual machine to define filter expressions and for efficiency these filters are either interpreted or compiled *on-the-fly* (during execution) or *just-in-time* (before execution). This virtual machine also enforces certain security requirements by inserting additional run-time checks. We enable efficient filter evaluation by dividing a filter into two parts. The first part, the *condition*, does not have any side effects and is used to determine which event to dispatch and which filter is to be executed. The second part, the *action*, provides a code sequence that is executed when its condition matches. The action part may have side effects. The condition part is organized in such a way that it allows a tree based evaluation to find the matching condition efficiently. Synchronizing access to local state is straightforward and the virtual machine will enforce this. To synchronize access to user state, the virtual machine contains lock instructions which map onto our thread package's shared lock mechanism.

In our system, active filters are implemented by a separate module that is used to demultiplex incoming events. Other systems, which are the source for events such as a network adapters, can implement this active filter mechanism directly. An example of an application for our system is shown in Figure 4.12. In this picture an incoming event is matched with the filters in the filter table and if one of the conditions matches the corresponding action is executed. The filter expressions in this example are denoted as pseudo expressions where the condition is the left hand side, followed by an arrow as separator, and the action is on the right hand side. The actual implementation is discussed in the next section.

In this example we have three different servers (*A*, *B*, and *C*) over which we balance the work load. We use a simple partitioning scheme where each server is given a proportional share of the work. Which server will get the request is determined by evaluating the filter conditions. For server *A* the condition is $U_A[workload] \leq U_A[total]/3$, meaning that its work load should be less than or equal to one third of the total work load of all servers in order for it to become true. In this pseudo expression, $U_A[workload]$ is a memory reference to offset *workload* in the virtual memory range shared with server *A* (*i.e.*, server's *A* private data). Similarly, $U_A[total]/3$ refers to the proportional share of all requests in progress by the servers.† When an event is dispatched to the filter module it evaluates the filter conditions to determine which filter expression applies and then executes the corresponding filter action. Unlike the condition, an action is allowed to make changes to the user and local state. In our example, the action consists of $U_A[workload]$++; $U_A[total]$++; *dispatch*" meaning that

†The filter expressions are limited to accessing local data and small portion of the address space of the user that installed the filter. In order for it to access global state, such as the *total* variable in this example which is shared over multiple servers, we use an aliasing technique. That is, all servers share a common physical page and agree on the offset used within that page. This page is then mapped into each server's protection domain. Each server makes sure that the filter module can access this aliased page. This is further discussed in Section 4.3.2.

it updates the per server work load, the total request count, and dispatches the associated event. As soon as the request has been processed, the server will decrease these variables to indicate that the work has been completed.

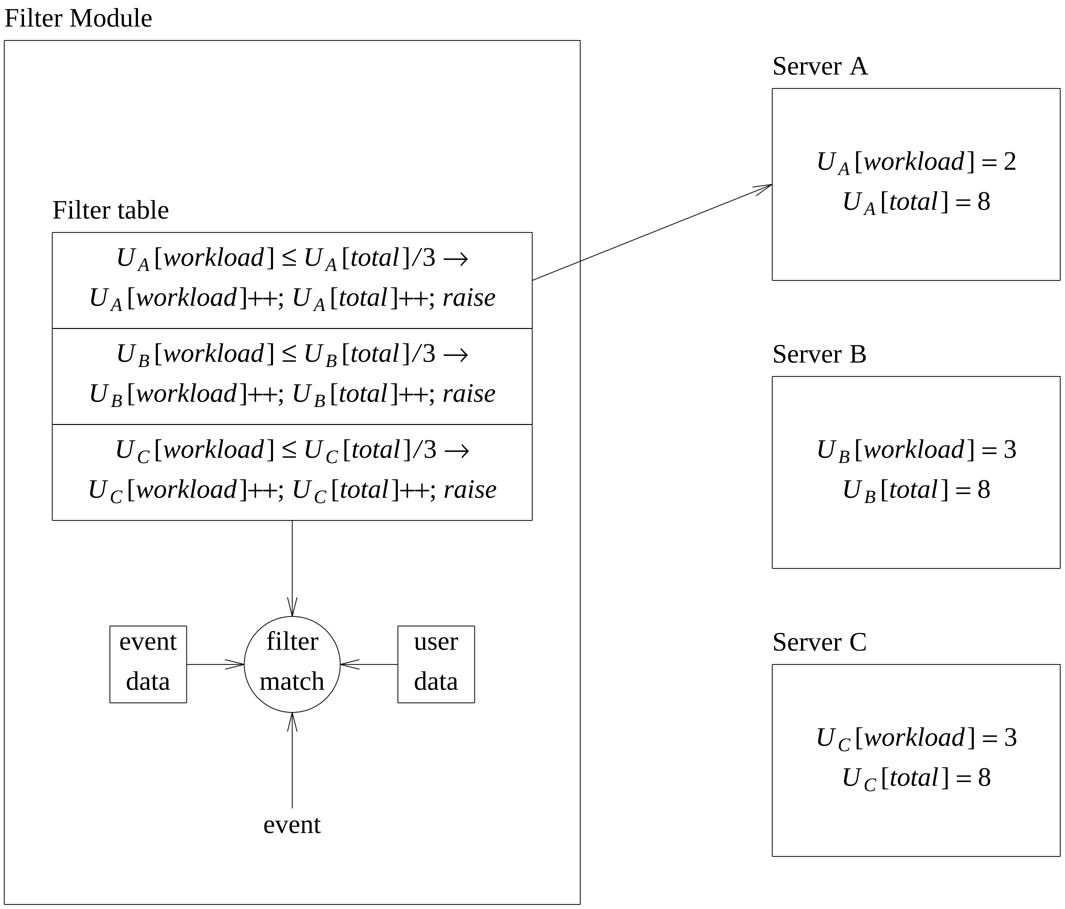

Filter Module
Filter table
$U_A[workload] \le U_A[total]/3 \rightarrow$
$U_A[workload]$++; $U_A[total]$++; *raise*
$U_B[workload] \le U_B[total]/3 \rightarrow$
$U_B[workload]$++; $U_B[total]$++; *raise*
$U_C[workload] \le U_C[total]/3 \rightarrow$
$U_C[workload]$++; $U_C[total]$++; *raise*
event data
filter match
user data
event
Server A
$U_A[workload] = 2$
$U_A[total] = 8$
Server B
$U_B[workload] = 3$
$U_B[total] = 8$
Server C
$U_C[workload] = 3$
$U_C[total] = 8$

Figure 4.12. Example of a load balancing filter.

It is interesting to note that the set of problems listed for active filters is similar to those of kernel extensions in Chapter 3. There we chose to use a different approach: signed binaries. The reason for this was that we wanted to keep the kernel small and only include those services required for the base integrity of the system. In addition, portability across multiple devices is not a major concern for kernel extensions. Still, with filters we explore a different kind of extension mechanism, one that is similar to application specific handlers in the ExOS kernel [Engler *et al.*, 1994].

In the next section we will discuss the design of the filter virtual machine and how to implement it efficiently. The section after that contains a number of sample applications for our filter mechanism.

4.3.1. Filter Virtual Machine

An active filter expression consists of two parts, a condition part which cannot have side effects and an action part which is essentially unrestricted filter code. In addition to these two expressions, our event dispatching service also associates a virtual memory range with a filter. This range corresponds to a virtual memory region in the user's address space to which the filter has access. This allows the filter to manipulate selected user data structures when the filter module is located in a different protection domain.

As pointed out in the previous section, portability, security, efficient evaluation and synchronization are key issues for active filters. Rather than denoting the filter expressions in pseudo-code as in the previous section, we express them as virtual machine instructions which are interpreted or compiled by the filter module. This approach is similar to the Java virtual machine (JVM) approach with the difference that the virtual machine is a RISC type machine which is based on Engler's VCODE [Engler, 1996], a very fast dynamic code generation system. This system is portable over different platforms and secure in the sense that the virtual machine enforces memory and control safety (see Section 3.3). Efficiency is achieved by using run-time compilation and optimization techniques to turn the virtual machine code into native code. Because the RISC type virtual machine is a natural match with the underlying hardware. code generation can be done faster and more efficiently than for stack based virtual machine such as Java bytecodes. To provide synchronization support we have extended the virtual machine to include synchronization primitives.

To implement filters more efficiently we have separated filter expressions into a condition and an action part and placed certain restrictions on conditions. This separation corresponds to the natural structure of first determining whether a filter applies before executing it. The restrictions placed on the condition part are: the expression should be *referentially transparent* (*i.e.*, should not have side effects), and have a sequential execution control flow (*i.e.*, no back jumps), These two restrictions allow the condition expression to be represented as a simple evaluation tree [Aho *et al.*, 1986]. Using this tree, we can construct a single evaluation tree for all filter conditions and apply optimization techniques such as common subexpression elimination to simplify the tree. To detect whether a condition has been evaluated we add marker nodes, denoting which condition matched, and bias the tree such that the first filter is the first to match (for simplicity we assume that only one filter can match, this restriction can be lifted by continuing the evaluation after a match). This evaluation tree can either be interpreted or, using dynamic run-time compilation techniques, be compiled into native code. In the latter case the evaluation tree is compiled each time a new filter condition is added rather than at first use-time as is the case with just-in-time compilers. As soon as a marker is reached the associated action expression is executed. Using an condition evaluation tree is especially advantageous when a large number of filters are used since a tree based search reduces the search time from $O(n)$ to $O(log\ n)$. The action expression, unlike the condition expression, does not have any restrictions placed on it.

The condition and action part of an active filter consist of a sequence of filter virtual machine instructions which are summarized on Figure 4.13. The filter virtual machine is modeled after a load-store RISC machine and the instruction set is intentionally kept simple to allow efficient run-time code generation. The registers of the filter virtual machine are 64-bits wide and the instructions can operate on different integer data types. These are quad, word, half word, and byte, and correspond to 64, 32, 16, and 8 bit quantities, respectively. The instructions themselves are divided into multiple groups as follows:

- *Binary operations.* These are the traditional binary operations such as addition, subtraction, bitwise exclusive or, and bit shift operations. The format of these instruction is a three tuple opcode: two operand registers and a result register.

- *Unary operations.* These are the traditional unary operations such as bitwise complement, register move, and type conversion. The format of these instructions is a two tuple opcode: the source and destination registers.

- *Memory operations.* These are the only operations that can load and store values to memory, and they are separated into two groups. The load and store user (ldu/stu) operations operate on the virtual memory range that was given by the user who also provided the filter expression. This allows filter expressions to manipulate user data structures during filter evaluation. The load and store local (ldl/stl) operations operate on the arguments associated with the event that caused the filter evaluation. These arguments contain, for example, the data of a network packet. The format of the load memory instructions is a three tuple opcode where the first operand is the source register, the second operand is the an offset, and the third operand is the result register.

- *Control of transfer operations.* The control transfer operations are subdivided into two groups: conditional and unconditional control transfers. The former group evaluates a condition (*e.g.*, less than, less or equal than, or greater than) and if the condition holds, control is transfered to the target address. The latter group transfers control unconditionally. This group also includes a link and return instruction which acts as procedure call and return. The format for conditional control transfer instructions is a three tuple opcode. The first two operand are the left and right hand side of the condition and the third operand is the target address. The jmp instruction's format is a single tuple opcode, its single operand is the target address. The lnk instruction is similar to jump except that it leaves the address of the next instruction after the link in the first operand register.

- *Procedure operations.* These operations assist the virtual machine in allocating the persistent and temporary register storage requirements for each pro-

Instruction	Operands	Type	Comments
add	rs1,rs2,rd	q,w,h,b	Addition
sub	rs1,rs2,rd	q,w,h,b	Subtraction
mul/imul	rs1,rs2,rd	q,w,h,b	Multiply
div/idiv	rs1,rs2,rd	q,w,h,b	Divide
mod	rs1,rs2,rd	q,w,h,b	Modulus
and	rs1,rs2,rd	q,w,h,b	Bitwise and
or	rs1,rs2,rd	q,w,h,b	Bitwise or
xor	rs1,rs2,rd	q,w,h,b	Bitwise xor
shl	rs1,rs2,rd	q,w,h,b	Shift left
shr	rs1,rs2,rd	q,w,h,b	Shift right
com	rs,rd	q,w,h,b	Bitwise complement
not	rs,rd	q,w,h,b	Bitwise not
mov	rs,rd	q,w,h,b	Register move
neg	rs,rd	q,w,h,b	Negate
cst	rs,rd	q,w,h,b	Load a constant
cvb	rs,rd	b	Convert byte
cvh	rs,rd	h	Convert half word
cvw	rs,rd	w	Convert word
ldu	rs,offset,rd	q,w,h,b	Load from user
stu	rs,rd,offset	q,w,h,b	Store to user
ldl	rs,offset,rd	q,w,h,b	Load from local
stl	rs,rd,offset	q,w,h,b	Store to local
blt	rs1,rs2,addr	w,h,b	Branch if less then
ble	rs1,rs2,addr	w,h,b	Branch if less then or equal
bge	rs1,rs2,addr	w,h,b	Branch if greater then or equal
beq	rs1,rs2,addr	w,h,b	Branch if equal
bne	rs1,rs2,addr	w,h,b	Branch if not equal
jmp	addr		Jump direct or indirect to location
lnk	rd, addr	w	Link and jump direct or indirect to location
enter	npr,ntr		Function prologue
leave			Function epilogue
raise	rs	q	Raise an event
lck	rs	q	Lock mutex
unlck	rs	q	Unlock mutex

Figure 4.13. Summary of the active filter virtual machine instructions. The types q, w, h, b correspond to unsigned 64, 32, 16, and 8 bit quantities, respectively, and rs and rd denote source and destination registers.

cedure. The enter instruction allocates the storage space, with the number of persistent and temporary registers as parameters, and *leave* will release it.

- *Extension operations.* The last group of instructions provides support for Paramecium specific operations. These include raising an event and acquiring and releasing locks. Raising an event can be used to propagate an event by a filter, as in the load balancing example in the previous section, and the synchronization primitives are used to prevent race conditions when accessing shared resources. The format of these instructions is a single tuple opcode. The operand contains the resource identifier for the event or lock variable.

The virtual machine has been designed to allow fast run-time code generation. As a result the instructions closely match those found on modern architectures. The run-time code generation process consists of two steps: the first is register allocation and the second is the generation of native instructions. The latter uses a straightforward template matching technique [Massalin, 1992], while the former is much harder as it consists of mapping the virtual machine registers onto the native registers. To accommodate fast register allocation we have divided the register set into two groups: those persistent across function calls and temporary registers. As suggested by Engler [Engler, 1996] the register allocator uses this order to prioritize the allocation. First it maps persistent registers and then the temporary registers to the native registers. This technique works quite well in practice, since modern RISC architectures tend to have many registers.

The filter virtual machine has a relatively traditional architecture with some exceptions. Unlike other filter systems, our system allows access to user data during the evaluation of a filter. This enables a number of applications which are discussed in the next section. It also raises synchronization issues since multiple threads may access the data concurrently. For this reason we augmented the virtual machine and added synchronization primitives to the basic instruction set that directly map onto our thread package's synchronization primitives. The shared synchronization state mechanism provided by our thread package ensures an efficient lock implementation. We purposely separated user memory accesses and synchronization to allow for flexible locking policies.

Another addition to the virtual machine is the ability to raise events. This is used by the action expression of a filter to propagate the event that caused the filter evaluation and is the foundation of our event dispatching mechanism. Embedding the event invocation in the action expression provides additional flexibility in that it is up to the action expression to determine when to raise the event. For example, a network protocol stack could implement an action expression on an intelligent I/O processor that handles the normal case. Exceptional cases, such as error handling or out of order processing, could be handled by the host processor which would be signaled by an event raised on the I/O processor.

4.3.2. Example Applications

Active filters provide a way to safely migrate computations to a different protection domain, including the kernel, and into intelligent I/O devices. The main purpose of active filters is to demultiplex events, but they can also be used to perform autonomous computations without dispatching the event. To illustrate the versatility of active filters we describe three different applications that take advantage of them. The first application elaborates on the load balancing example in Section 4.3. The next example shows how to build a distributed shared memory system using active filters. In the last example we discuss the use of active filters in intelligent I/O devices and discuss some of its design challenges.

Server Load Balancing

The server load balancing example from Section 4.3 is an example where active filters are used to select a recipient of an event based on the workload of a server. A typical application of this service would be web server load balancing, where the active filter module is part of the network protocol stack and the web servers reside in different protection domains. In this section we will illustrate the example further by discussing some of the missing details.

In order for the web server to receive events it has to register an active filter with the demultiplexing module in the network protocol stack. These filters are described in terms of the filter virtual machine and the condition expression for our load balancing example is shown in Figure 4.14. This condition evaluates to true (1) when the condition matches, else it evaluates to false (0). In this example, the condition expression only operates on temporary registers and accesses the user variables workload and total. These are the variables shared between the server and the filter expressions. By convention the result of a condition is returned in temporary register zero.

```
        enter   0,3             / start procedure
        ldu     0,workload,t1   / t1 := U_A[workload]
        ldu     0,total,t2      / t2 := U_A[total]
        div     t2,3,t2         / t2 := U_A[total]/3
        cst     1,t0            / t0 := true
        ble     t1,t2,lesseq    / U_A[workload] ≤ U_A[total]/3
        cst     0,t0            / t0 := false
lesseq:
        leave                   / end procedure
```

Figure 4.14. Filter virtual machine instructions for the $U_A[workload] \le U_A[total]/3$ condition.

A problem that arises in this condition is that filter expressions can only access a portion of the web server's address space. That is, the virtual memory range the web server passed on to the active filter module when registering the filter. Other memory is off limits including virtual memory ranges used by other active filters. To overcome this limitation we use page aliasing for sharing the global variable total. While workload is private to each server, the total variable is shared among several servers and represents the total number of jobs in progress. Before registering the active filters, the servers agree on a single shared page to hold this total variable and each server makes it available in the virtual memory range associated with the active filter. That is, the memory range which the filter module, and consequently the filter, can access.

The condition expression operates on shared variables and is vulnerable to race conditions caused by concurrent accesses to the variables. In this case, however, these are harmless and cause at most a transient unbalanced load. For the action expression shown in Figure 4.15, proper locking is crucial since it modifies the shared variables. For this the action expression acquires a mutex before updating the shared variables. It releases the mutex before raising the event to propagate the event invocation to the server. It is up to the server to properly decrease these values when it has processed the request.

```
enter   0,1              / start procedure
lck     mutex            / acquire shared mutex
ldu     0,workload,t0    / t0 := $U_A[workload]$
add     t0,1,t0          / t0 := t0 + 1
stu     t0,0,workload    / $U_A[workload]$ := t0
ldu     0,total,t0       / t0 := $U_A[total]$
add     t0,1,t0          / t0 := t0 + 1
stu     t0,0,total       / $U_A[total]$ := t0
unlck   mutex            / release shared mutex
raise   event            / demultiplex event
leave                    / end procedure
```

Figure 4.15. Filter virtual machine instructions for the $U_B[workload]$++; $U_B[total]$++; *raise* action.

Distributed Shared Memory

Another example of the use of active filters is their application in parallel programs that run on a collection of loosely coupled machines, such as a collection of workstations (COW). These parallel programs are usually limited by the communication latency between different machines and would benefit from latency reduction. One way of reducing the latency is to migrate part of the computation into the device driver's address space where it could inspect and process each incoming packet before passing it to the application. In fact, latency can be even further reduced by moving part of the computation, the processing of simple packets, into an intelligent I/O device which avoids interrupting the kernel entirely.

Typical candidates that might benefit from this approach are the traditional *branch-and-bound* algorithms [Bal, 1989] that solve problems such as the traveling salesman problem (TSP). A TSP solving program could use message passing to broadcast its current bound to the group of cooperating processors. These processors could use the active filters to determine whether the message was intended for them and take the bound from the network message and assign it to the bound variable shared with the user process running the application. The TSP solving application would periodically examine the current bound and adjust its search accordingly. By moving this functionality into an intelligent network I/O adapter we can avoid interrupting the main processor all together, but this raises a number of issues that are discussed in the next example.

Predicate Addressing

Our last example is a predicate addressing scheme where incoming network packets are selected based on predicates rather than fixed addresses such as a hardware MAC address. In such a system the user typically supplies a predicate filter which is installed and implemented on an intelligent I/O device. These predicates can be used, for example, to implement cache consistency by having the user register predicates that match the objects in its cache. Just as with snooping cache protocols [Handy, 1993], when the user sees an update for a cached object the update will invalidate the user's copy†. Not surprisingly, active filters model this concept of predicate addresses quite naturally since it was one of the original ideas behind active filters. In this subsection we will discuss the implications of migrating active filter computations to an intelligent I/O device and use predicate addressing as the example.

The main goal of predicate addresses is to reduce the workload for the host processor by providing finer grained control over the accepted packets that interrupt the host processor. These interruptions can be further reduced by migrating part of the user computation into an intelligent I/O device. A typical intelligent network I/O adapter consists of a network interface, a bus interface, a general purpose CPU, memory, and possibly additional hardware support for encryption, check summing, and memory management. The issues involved in implementing active filters on an intelligent I/O device are similar to the generic active filter issues. These are: portability, security, efficiency, and synchronized access to local and user state. The solutions are also similar, except for security and synchronized access to user state, which have to be handled differently.

The security issues are different in that an intelligent I/O device can typically perform bus master I/O. That is, it can read and modify any physical memory location in main memory without assistance or approval from either the main processor of the MMU. Consequently, any program running on them can read and modify any memory location. To prevent this we can either resort to sandboxing every virtual machine instruction that accesses main memory or employ a proper MMU on the I/O bus. As in described in Section 3.3, sandboxing has a nonnegligible performance impact for gen-

†Predicate addressing assumes that the physical interconnect is a broadcast medium such as Ethernet or token ring network. Efficiently implementing predicate addressing on nonbroadcast networks, such as ATM and gigabit Ethernet, can be done by routing messages based predicate unions.

eric processors, but in this case it might be a viable solution. Our virtual machine is sufficiently simple and bus master transfers for small sizes, such as a 4-byte integer, are sufficiently expensive that the sandboxing overhead might be insignificant. The other solution is to add a separate MMU on the I/O bus as described in 3.4.6.

Synchronization between the intelligent I/O device and the main processor should occur either through hardware semaphores when they are available or by implementing, for example, Dekker's Algorithm [Ben−Ari, 1990]. This choice depends on the atomicity properties of the device and host processor memory accesses.

The advantage of using a filter virtual machine is that the filter expressions can also be implement on different hardware devices. For example, the intelligent I/O device could be equipped with *field programmable gate arrays* (FPGA). Assuming that filter condition expressions do not change often, we can compile them into a netlist and program the FPGA to perform parallel filter matching. A different approach would be to use special purpose processors that are optimized to handle tree searches (pico processors).

4.4. Discussion and Comparison

Threads have a long history. The notion of a thread as a flow of control dates back to the Berkeley time-sharing system [Lampson *et al.*, 1966] from 1966. Back then they were called processes. These processes interacted through shared variables and semaphores [Dijkstra, 1968a]. The programming language PL/1, also dated back to 1965, contained a CALL start (A, B) TASK construction which would call the function start as a separate task under OS/360 MVT. Reportedly, a user-level thread package for Multics was written around 1970 but never properly documented. It used multiple stacks in a single heavyweight process. With the advent of UNIX in the early 1970s, the notion of multiple threads per address space disappeared until the appearance of the first microkernels in the late 1970s and early 1980s (V [Cheriton, 1988], Amoeba [Tanenbaum *et al.*, 1991], Chorus [Rozier *et al.*, 1988], Accent [Fitzgerald and Rashid, 1986], and Mach [Accetta *et al.*, 1986]). Nowadays, multiple threads per address space are found in most modern operating systems (Solaris [Vahalla, 1996], QNX [Hildebrand, 1992], and Windows NT [Custer, 1993]).

Active messages date back to Spector's remote operations [Spector, 1982]. These operations would either fetch or set remote memory by sending the remote host simple memory operations. Von Eicken extended this concept by replacing the function descriptor with the address of a remote function. Upon receipt of this active message the recipient would, directly from the interrupt handler, execute the designated remote function [Von Eicken *et al.*, 1992]. This approach takes the model of the network as an extension of the machine's internal data bus and achieved very low latency communication at the expense of the security of the system.

The traditional active message mechanism has a number of serious drawbacks associated with it in the form of security and synchronization. Some of these were solved in our implementation for Amoeba [Van Doorn and Tanenbaum, 1994] but it still restricted the handler in that it could only synchronize on a single shared lock. Wallach *et al.* devised a technique called optimistic active messages (OAM) [Wallach *et al.*, 1995] which enabled the handler to consist of general purpose code while still performing as well as the original active message implementation. They delayed the creation of the actual thread until the handler was either blocked or took too long to execute.

Thread migration and synchronization state sharing are the techniques used to support Paramecium's lightweight protection model where an application is subdivided into multiple protection domains. Thread migration is an obvious communication mechanism to transfer control between two closely intertwined protection domains and it can be implemented efficiently with some minimal operating system support. Thread migration is also used by Ford *et al.* to optimize IPCs between two protection domains [Ford and Lepreau, 1994]. Other systems that included migrating threads are Clouds [Dasgupta and Ananthanarayanan, 1991] and Spring [Mitchell *et al.*, 1994]. Some authors, especially those working on loosely coupled parallel systems, use the term thread migration to refer to RPC-like systems where a thread logically migrates from one processor to another [Dimitrov and Rego, 1998; Mascaranhas and Rego, 1996; Thitikamol and Keleher, 1999]. This is different from our system where the thread migrates into a different protection domain while remaining a single schedulable entity.

Synchronization state sharing is a novel aspect of our thread system that finds its roots in a commonly used optimization technique for kernel-level thread systems. Rather than calling the kernel each time a mutex is grabbed, a local copy is tried first and the kernel is only called when there is contention for the lock. One system that has implemented this optimization is Amoeba [Tanenbaum *et al.*, 1991]. In our system we have extended this mechanism to make it an integral part of the thread system and enabled multiple address spaces to share the same lock efficiently. The latter is important since threads can migrate from one protection domain to another and multiple protection domains can work on the same shared data. Our synchronization state sharing technique could be used to optimize Unix inter-process locking and shared memory mechanisms [IEEE, 1996].

TCP/IP [Stevens, 1994] has a long history too that dates back to a DARPA project in the late 1960's and nowadays forms the foundation for the Internet. Our TCP/IP network protocol stack was implemented to provide network connectivity for Paramecium and to demonstrate the use of pop-up threads and the versatility of our extensible operating system by implementing an efficient cross-domain data sharing mechanism. Our shared buffer scheme is in many respects similar to Pai's work on IO-Lite [Pai *et al.*, 2000]. This is not surprising since both are inspired by earlier work from Druschel on cross-domain data transfer [Druschel and Peterson, 1993].

IO-Lite is a unified buffer scheme that has been implemented in FreeBSD [McKusick *et al.*, 1996]. It provides immutable buffers that may only be initialized by the initial producer of the data. These immutable data buffers are kept in aggregates, tuples of address and length pairs, which are mutable. Adding extra data to an IO-Lite buffer consists of creating a new immutable data buffer and adding a tuple for it to the aggregate list. To transfer a buffer from one protection domain to another it suffices to pass the aggregate for it. When passing an aggregate, the kernel will ensure that the immutable data buffers are mapped read-only into the recipient's virtual address space. Implementing IO-Lite on Paramecium would be relatively straightforward.

Instead, we designed a different shared buffer scheme, where the sharing granularity is a buffer pool. In our scheme data buffers are mutable and therefore access control on the buffer pool is much more important. Once a shared buffer is created and every party has successfully bound to it, which causes the memory to be mapped into the appropriate protection domains, data is passed by passing an offset into the shared buffer space. The fact that buffers are mutable and shared among multiple parties requires extra care in allocating buffers. For example, for buffers traversing up the protocol stack it is important that a higher layer does not influence the correctness of its lower layers. The converse is true for buffers traversing down the protocol stack. This is easily achieved by using different buffer pools.

The TCP/IP implementation for Paramecium is based on the Xinu protocol stack [Comer and Stevens, 1994] with some modifications taken from the BSD network stack [McKusick *et al.*, 1996]. We heavily modified the stack to use the shared buffer scheme described above, to make it multithreaded, and to take advantage of our pop-up threads. The pop-up threads are used to shepherd incoming network messages through the protocol stack to the user application. This is similar to the packet processing ideas found in the X-kernel [Hutchinson *et al.*, 1989]. Since the protocol stack we based our work on was written as a teaching tool, the performance is poor. We did not attempt to improve this, since we used it only to demonstrate our system and provide remote login capabilities.

Active filters are an efficient and flexible event demultiplexing technique. The filter consists of two components, a condition part and an action part. The condition part establishes whether to execute the associated action part and in order to implement condition expressions efficiently we place certain restrictions on it. These restrictions allow the efficient evaluation of filter expressions. Both filter condition and action expressions are written in virtual machine instructions which can be dynamically compiled at run time. We used the virtual machine approach, as opposed to code signing, to achieve portability, security, and efficiency. The virtual machine is based on VCODE, a very fast dynamic code generation system [Engler, 1996]. We have extended it to include safe access to user data structures, added synchronization, and optimized it for condition matching. With it we have explored some sample applications that use active filters for load balancing, distributed shared memory, and predicate addressing.

Active filters are in some sense a different extension technique to the one we used to extend our kernel. They are more akin to ExOS's application specific handlers (ASH) [Engler *et al.*, 1994] and their application in a dynamic packet filter (DPF) [Engler and Kaashoek, 1996]. Just as with ASHes, active filters allow migration of computation into the kernel and user address spaces, but active filters are also designed to migrate to devices such intelligent I/O devices or even FPGAs. The primary use of active filters is for event demultiplexing, for this we have incorporated an efficient filter selection mechanism that can be used for demultiplexing events based on arbitrary filter expressions.

Active filters have a number of interesting applications, besides the ones mentioned in Section 4.3.2 they can also be used for system call argument passing as in Pebble [Gabber *et al.*, 1999]. One of the most interesting examples of active filters is the ability to safely migrate computations into an intelligent I/O device. This is different from U-Net where the network device is safely virtualized [Von Eicken *et al.*, 1995]. Our work is more closely related to Rosu's *Virtual Communication Machine* (VCM) architecture [Rosu *et al.*, 1998] where, as in our system, applications have direct access to the network I/O device and the VCM enforces protection and communicates with the host application using shared memory segments.

5

Run Time Systems

In this chapter we discuss the design of two major applications for Paramecium, which demonstrates its strength as an extensible operating system. The first application is an extensible run-time system for Orca, a programming language for parallel programming. For this system the emphasis is on providing a frame work for application specific optimizations and show examples where we weaken the ordering requirements for individual shared objects, something that is quite hard to do in the current system.

The second application consists of a secure Java™ Virtual Machine. The Java Virtual Machine is viewed by many as inherently insecure despite all the efforts to improve its security. In this chapter we describe our approach to Java security and discuss the design and implementation of a system that provides operating system style protection for Java code. We use Paramecium's lightweight protection domain model, that is, hardware separated protection domains, to isolate Java classes and provide: access control on cross domain method invocations, efficient data sharing between protection domains, and memory and CPU resource control. Apart from the performance impact, these security measures, when they do not violate the policy, are all transparent to the Java programs. This is even true when a subclass is in one domain and its super class is in another. To reduce the performance impact we group classes and share them between protection domains and map data in a lazy manner as it is being shared.

The main thesis contributions in this chapter are an extensible run-time system for parallel programming and a secure Java virtual machine. Our secure Java virtual machine, in particular, contains many subcontributions: adding hardware fault isolation to a tightly coupled language, a new run-time data relocation technique, and a new garbage collection algorithm for collecting memory over multiple protection domains while taking into account sharing and security properties.

5.1. Extensible Run Time System for Orca

Orca [Bal *et al.*, 1992] is a programming language based on the *shared object model*, which is an object-based shared memory abstraction. In this model the user has the view of sharing an object among parallel processes and with multiple parties invoking methods on it. It is the task of the underlying run-time system to preserve consistency and implement this view efficiently. The current run-time system guarantees *sequential consistency* [Lamport, 1979] for shared object updates. The shared objects are implemented by either fully replicating the shared object state or by maintaining a single copy. This trade-off depends on the read/write ratio of the shared object. If a shared object has a higher read ratio it is better to replicate the state among every party so that reads are local and therefore fast. Writes use a global consistent update protocol which is therefore slower. When a shared object has a higher write ratio, keeping a single copy of the shared object state is more efficient since it reduces the cost of making a globally consistent update. Current Orca run-time systems implement this scheme and dynamically adjust the shared object state distribution at run time [Bal *et al.*, 1996].

The Orca system includes a compiler and a run-time system. The run-time system uses I/O, threads, marshaling, group communication, message passing, and RPC to implement the shared objects and many of its optimizations. Current run-time systems implement a number of these components and rely on the underlying operating system to provide the rest. In a sense, the current run-time system is *statically configurable* in that it requires rebuilding and some redesigning at the lowest layers when it is ported to a new platform or when support is added for a new device.

In FlexRTS [Van Doorn and Tanenbaum, 1997] we enhanced the Orca run-time system to take advantage of Paramecium’s extensibility mechanisms. The ability to dynamically load components enabled us to specify new or enhanced implementations at run time. Combined with the ability to load these implementations securely into the kernel it is possible to build highly tuned and application specific run-time systems. The advantages of using a Paramecium-based run-time system over the existing system are: performance enhancements, debugging, tracing, and the ability to create application specific implementations for individual Orca objects.

In a sense, FlexRTS shares the same philosophy as the Paramecium kernel: it starts with a minimal run-time system which is extended dynamically on demand. Unlike the kernel, where the guideline was to remove everything from the kernel that was not necessary to maintain the integrity of the system, FlexRTS follows the guideline of removing everything from the run-time system that dictated a particular implementation. That is, the base run-time system does not include a component that imposes a certain implementation, such as a group communication protocol or a thread package.

The key advantage of FlexRTS is the ability to provide application specific implementations for individual shared objects and control their environment. We control a shared object’s implementation by instantiating its implementation in a programmer defined place in the Paramecium per-process name space and using the name space search rules which were described in Chapter 2. For example in Figure 5.1 we have a component called /program/shared/minimum, representing a shared integer object. This shared integer implementation requires a datagram service for communicating with other instances of this shared integer object located on different machines. By associating a search path with the component name, we can control which datagram service, registered under the predefined name datagram, it will use. In this case /program/shared/minimum will use /program/datagram. When no search path is associated with a given name, its parent name is used recursively up to the root until a search path is found. This allows us to control groups of components by placing an overriding name higher in the hierarchy.

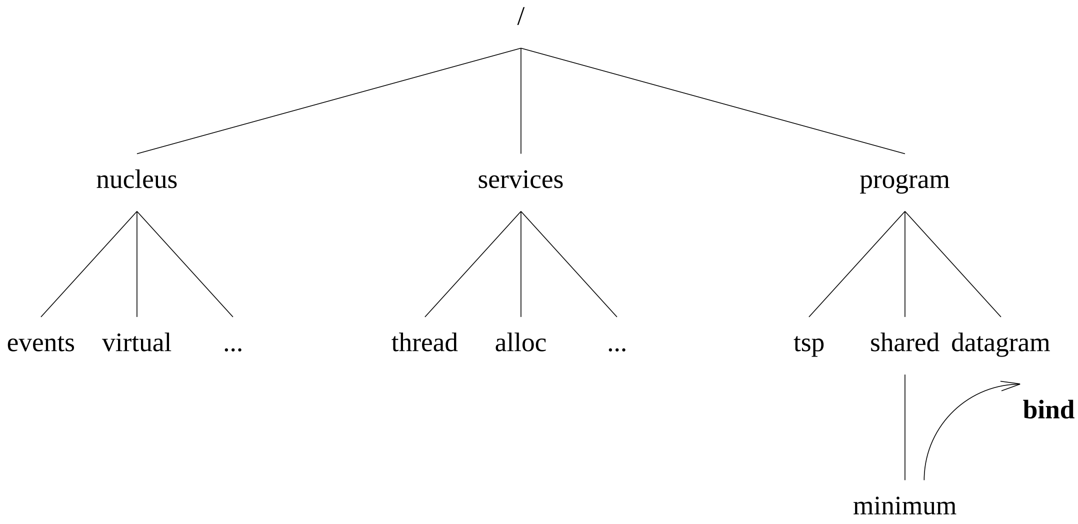


Figure 5.1. Example of controlling a shared object binding to a name.

The advantage of controlling user level components at binding time is the ability to provide different implementations that include performance improvements or debugging code. Individual shared object implementations can use different marshaling routines, different network protocols, different networks, debugging versions, *etc*. On machines where the context switch costs are high, all of the protocol stacks and even drivers for nonshared devices can be loaded into the run-time system to improve its performance. This scheme can also be used to reduce the copying of packets [Bal *et al.*, 1997].

Placing components into the kernel is useful for performance improvements and availability. The performance improvements are the result of a reduced number of context switches and the direct access to devices which are shared among other processes. Drivers for these cannot be loaded into user space.

On time-shared systems it is often useful to place services that are performance bottle-necks in the kernel for availability reasons. These are always runnable and usually do not get paged out, even under a high load. For example, consider a collection of workstations computing on a parallel problem with a job queue. The job queue is a perfect candidate to be down-loaded into the kernel. Requests for new work from a process on a different machine would then be dealt with immediately without having to wait for the process owning the job queue to be paged in or scheduled.

Hybrid situations, where part of the component is in the kernel and part in user space, are also possible. For example, consider the thread package on our implementation platform. Because of the SPARC architecture each thread switch requires a trap into the kernel to save the current register window set. To amortize this cost we instantiate the thread package scheduler in the kernel, and use synchronization state sharing to allow fast thread synchronization from user and kernel space. Although possible, it is undesirable to load the whole program into the kernel. It is important for time-sharing and distributed systems to maintain some basis of system integrity that, for example, can be used to talk to file servers or reset machines. Adding new components to the kernel should be done sparingly.

Our FlexRTS run-time system is implemented as a component according to our object model (see Chapter 2). It consists of an Orca class which exports the standard object interface, the standard class interface, and the Orca process interface. This latter interface is used to create new processes, possibly on different processors. The Orca shared objects are instantiated by creating an instance of this Orca class. It is up to the run-time system to implement Orca's language semantics which for shared objects consists of providing sequential consistency. All though in some application specific cases these language semantics can be relaxed to provide more efficient shared object implementations.

The FlexRTS component exports two main interfaces. The first is the Orca process interface which is part of the class. It assists in creating new Orca processes and Orca shared objects. The latter are not created through the standard class interface since they require additional arguments. The interface for Orca shared objects is shown in Figure 5.2. It consists of reader-writer mutex like functions to signal the start and end of read and write operations (start_read, end_read, start_write, and end_write). These are performed in-line by compiler generated code if no synchronization is required. Otherwise, the dooperation method is used to invoke a method on the Orca object. The isshared method determines whether the object is shared or local. Before creating a new object the run-time system checks the instance name space to determine whether a specific implementation already exists. If an implementation is found and it exports the Orca shared object interface as described above, it is used instead of the one provided by the run-time system. This enables applications to use specific shared object instances.

Method	Description
localread = start_read()	Signal the object implementation that a read operation will be performed. When this function returns nonzero the read can be performed locally.
end_read()	Signals the end of a read operation.
localwrite = start_write()	Signal the object implementation that a write operation will be performed. When this function returns nonzero the write can be performed locally.
end_write()	Signals the end of a write operation.
shared = isshared()	Is this a shared Orca object?
dooperation(operation, arguments, results)	Perform an operation on the object. The kind of operation, read or write, is determined by the surrounding start and end method calls.

Figure 5.2. Orca shared object interface.

In the next subsections we discuss some extensions for our FlexRTS run-time system. These extensions include an object-based group active message protocol to provide efficient shared-object updates [Van Doorn and Tanenbaum, 1994] and a shared integer object that is implemented partially in the kernel and partially in the user's address space. We complete this section by some specific examples of how parallel programs can take advantage of optimizations provided by our extensible run-time system.

5.1.1. Object-based Group Active Messages

The performance of parallel programming systems on loosely-coupled machines is mainly limited by the efficiency of its message passing communication architecture. Rendezvous and mailboxes are the traditional communication mechanisms upon which these systems are built. Unfortunately, both mechanisms incur a high latency at the receiver side between arrival and final delivery of the message.

An alternative mechanism, active messages [Von Eicken *et al.*, 1992], reduces this latency by integrating the message data directly into the user-level computation as soon as it arrives. The integration is done by a user specified handler, which is invoked as soon as possible after the hardware receives the message.

For interrupt-driven architectures the most obvious design choice is to run the handler directly in the interrupt service routine. This raises, however, a number of problems: protection, possibility of race conditions, and the possibility of starvation and deadlock. Consequently, the handler cannot contain arbitrary code or run indefinitely.

The shared data-object model [Bal, 1991] provides a powerful abstraction for simulating distributed shared memory. Instead of sharing memory locations, objects with user-defined interfaces are shared. Objects are updated by invoking operations via their interfaces. The details of how these updates are propagated are hidden by the implementation. All operations on an object are serialized.

Shared objects are implemented using active replication. To do this efficiently, we have implemented a group communication system using active messages. In our implementation, the run-time system associates a mutex and a number of regular and special operations with each object. These special operations are invoked by sending an active message. They are special in that they must not block, cause a protection violation, or take longer than a certain time interval. They are executed in the network interrupt handler and run to completion once started. This means that once they are executing they are never preempted by other active messages or user processes. Other incoming active messages are buffered by network hardware which may eventually start dropping messages when the buffers run out. Hence the need for a bounded execution time. When the mutex associated with an object is locked, all incoming active messages for it are queued and executed when the mutex is released. Therefore, active message operations do not need to acquire or release the object's mutex themselves since it is guaranteed to be unlocked at the moment the operation is started.

Active message invocations are multicast to each group member holding a replica of the shared-object. These multicasts are totally-ordered and atomic. That is, in the absence of processor failures and network partitions it is guaranteed that when one member receives the invocation, all the others will too, in exactly the same order.

Associating a lock with each object is necessary to prevent an active message from starting an operation while a user operation was already in progress. Active message operations are bounded in execution time to prevent deadlock and starvation. The restrictions placed on active message handler are currently expected to be enforced by the compiler (*i.e.,* no unbounded loops) and the run-time system. These assumptions may be relaxed by using pop-up threads as described in Section 4.1.

Each replica of the shared object is registered at the kernel under a unique object name together with an array of pointers to its operations. To perform a group active message operation, a member multicasts an invocation containing the object name, the operation to be performed (an index into the object's interface array), and optional arguments.

The multicasting is performed by a sequencer protocol, shown in Figure 5.3, that is akin to Amoeba's PB protocol [Kaashoek and Tanenbaum, 1991]. In a sequencer protocol, one member is assigned the special task of ordering all messages sent to the group. In order to send a message to the group, a member sends the message to the sequencer which will assign a unique and increasing sequence number to the message before multicasting it to the group. The main difference between our new protocol and the PB protocol is that in our protocol the individual members maintain the history of messages they sent themselves instead of the sequencer. This makes the sequencer as fast as possible since it doesn't have to keep any state. The sequencer could even be implemented as an active filter on the intelligent network interface card. Our implementation takes advantage of the underlying hardware multicasting capabilities. For efficiency reasons, we have limited the size of the arguments to fit in one message.

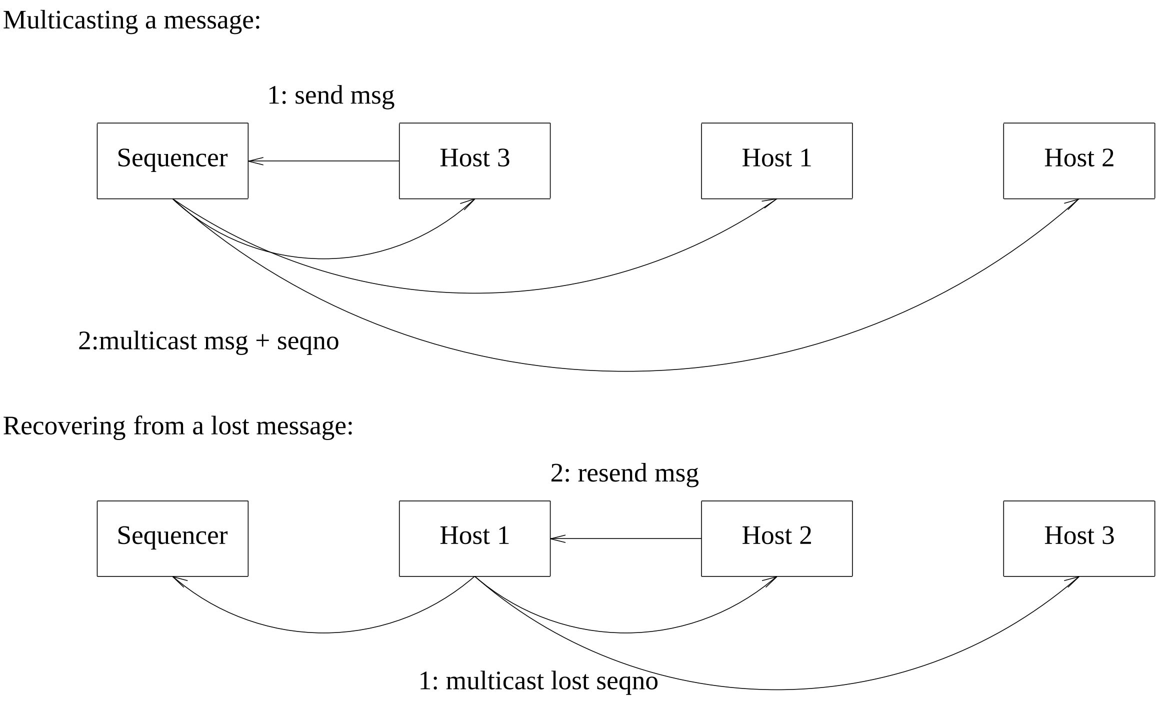


Figure 5.3. Group multicast with a sequencer.

Recovery of lost messages occurs when a member notices a gap in the sequence numbers of the received messages. In such a case, the member multicasts to the group that it has missed the message. The member that originated the message will send it again as a point-to-point message to the member that missed it.

When sending a message to the sequencer, the member includes the sequence number of the last message the member successfully delivered to the application. The sequencer uses this to determine the lower bound on the sequence numbers seen by every member and piggy backs it on every message. The members use this additional message data to purge their message queues since they do not have to remember messages received by every member. Silent members periodically send an information messages to the sequencer containing the sequence number of the last delivered message. This prevents silent members from filling up the message queues.

When a network packet containing an invocation arrives at a machine, it is dispatched to the active message protocol code. This code saves machine registers, examines device registers, and queues a software interrupt which, in turn, calls our protocol dispatcher. This routine does all of the protocol processing. Once it has established that the invocation is valid, it checks the associated mutex. If this mutex is locked, it queues the request on a per-object queue in FIFO order. If the mutex is not locked, the dispatch routine maps in the context of the process to which the object handler belongs and makes an up call to it. Whenever an object's mutex is released its lock queue is checked.

The main problems with active messages are related to the incoming message handler: can an active message operation block, how to prevent it from causing protection violations, and how long can it execute? In our current implementation active message handlers are user specified interrupt routines which should not block and should run to completion with a certain time interval. In our model these restrictions are expected to be enforced by the compiler and the run-time system.

A more general view is conceivable where user-level handlers have no limitations on the code or on the time to execute. One possible implementation is the use of continuations [Hsieh *et al.*, 1994] whenever a handler is about to block. However, with continuations it is hard to capture state information and dealing with exceeding execution quanta is tedious.

Another possible implementation is to create a pop-up thread for the active message handler (see Chapter 4 for a discussion). This pop-up thread is promoted to a real thread when it is about to block or when it runs out of time. The pop-up thread is created automatically by means of the processor's interrupt mechanism. Every network device has its own page of interrupt stack associated with it. Initially the handler executes on this stack and when it is turned into a real thread it inherits this stack and the network device's interrupt stack is replaced by a new page. This requires a reasonable number of preallocated interrupt stack pages. When we run out of these we drop incoming messages and rely on the active message protocol to recover.

5.1.2. Efficient Shared Object Invocations

To get some idea of the trade-offs and implementation issues in a flexible runtime system, consider the Orca shared object definition in Figure 5.4. This object implements a shared integer data object with operations to set a value, assign, return the value, value, and wait for a specific value, await. The methods from this object instance may be invoked from different, possibly distributed, Orca processes while maintaining global sequential consistency.

There are different ways to implement this shared integer object on Paramecium. They all depend on a trade-off between integrity, performance, and semantics. That is, if we implement the shared integer object in the kernel and consequently sacrifice some of the integrity of the system we can improve the performance by using active messages and eliminating cross protection domain calls. Likewise, if we relax the shared integer object semantics to unreliable PRAM consistency instead of sequential consistency, we can use a simple unreliable multicasting protocol instead of a heavyweight total ordered group communication protocol. Of course, these trade-offs do not always make sense. On a multiuser system, trading system integrity for performance is not an option, but on an application-specific operating system it might be. Even for

```
object specification IntObject;
  operation value(): integer;
  operation assign(v: integer);
  operation await(v: integer);
end;

object implementation IntObject;
  x: integer;

  # Return the current object value
  operation value(): integer
    begin return x end;

  # Assign a new value to the object
  operation assign(v: integer);
    begin x := v end;

  # Wait for the object to become equal to value v
  operation await(v: integer);
    begin guard x = v do od end;

begin x := 0 end;
```

Figure 5.4. An Orca shared integer object.

application-specific operating systems one would generally prefer the ability to debug a program over sacrificing system integrity.

In the remainder of this section we explore the design of a shared integer object that is implemented as a safe kernel extension. The idea behind this concept is that the shared integer kernel extension is among a set or toolbox of often used Orca shared object implementations which the programmer can instantiate at run time. The Orca program and run-time system itself still run as a normal user process, only the safe extensions are loaded into the kernel's address space. The goal of this work is to provide an Orca run-time system with a normal process failure model where some common shared object implementations may be loaded into the kernel at run-time.

Each method of this shared integer object implementation can be invoked remotely. For an efficient implementation we use a technique similar to optimistic active messages [Van Doorn and Tanenbaum, 1994; Von Eicken *et al.*, 1992; Wallach *et al.*, 1995]. When a message arrives, the intended object instance is looked up and the method is invoked directly from the interrupt handler. When the method is about to block on a mutex, it is turned into a regular thread.

To reduce the communication latency and provide higher availability for this shared object instance, we map its code read-only into both the kernel and user address spaces. This allows the methods to be invoked directly by kernel and possibly by the user. The latter depends on the placement of the instance state. Under some conditions the user can manipulate the state directly; others require a trap into the kernel. Obviously, mapping an implementation into the kernel requires it to be signed before hand.

In this simple example, mapping the object instance data as read/write in both user and kernel address space would suffice, but most objects require stricter control. To prevent undesired behavior by the trusted shared object implementation in the kernel, we map the object state as either read-only for the user and read-write for the kernel or vice versa, depending on the read/write ratio of its methods. For example, when the local (*i.e.*, user) read ratio is high and the remote write ratio is high, the instance state is mapped read/writable in the kernel and readable in the user address space. This enables fast invocation of the value and assign methods directly from kernel space (*i.e.*, active messages calls), and the value method from user space. In order for the user to invoke assign it has to trap to kernel space.

Another example of extending the kernel is that of implementing Orca guards. Guards have the property that they block the current thread until their condition, which depends on the object state, becomes true. In our example, a side effect of receiving an invocation for assign is to place the threads blocked on the guard on the run queue after their guard condition evaluated to true. In general the remote invoker tags the invocation with the guard number that is to be re-evaluated.

For our current run-time system we are hand-coding in C++ a set of often used shared object types (shared integers, job queues, barriers, etc). These implementations are verified, signed, and put in an object repository. For the moment, all our extensions and adaptations involve the thread and communication system, *i.e.*, low level services. These services provide call-back methods for registering handlers. For a really fast and specialized implementation, for example the network driver, one could consider integrating it with the shared object implementation.

5.1.3. Application Specific Optimizations

In this section we discuss some typical Orca applications and discuss how they can benefit from our flexible run-time system. These applications are: the traveling salesman problem, successive overrelaxation, and a 15-puzzle using the IDA^* tree search algorithm.

Traveling Salesman Problem

The traveling salesman problem (TSP) is the following classical problem [West, 1996]: given a finite number of cities and a cost of travel between each pair, determine the cheapest way of visiting all cities and returning to the starting city. In graph theory terms, the TSP problem consists of finding a Hamiltonian cycle in a directed graph with the minimum total weight.

In order to implement a parallel program that solves these problems we use a *branch-and-bound* type of algorithm. That is, the problem is represented as a tree with the starting city as the root. From the root a labeled edge exists to each city that can be reached from that root, this is applied recursively for each interior node. Eventually, this will lead to a tree representing the solution space with the starting city as the root and all the leaves, and each path from the root to the leaf represents a Hamilton tour. This tree is then searched but only solutions that are less than the bound are considered. If a solution that is less than the current bound is found it will become the new bound and further prunes the search space.

Parallelizing the TSP problem using this branch-and-bound algorithm is straightforward. Arbitrary portions of the search space can be forked off to separate search processes and it is only necessary to share the bound updates. The graph data itself is static and does not require sharing. An Orca implementation of TSP would implement the bound as a shared integer object [Bal, 1989] with all its sequential consistency guarantees. However, the TSP solving algorithm does not require such a strong ordering guarantee on the current bound. Instead a much weaker guarantee, such as unreliable and unordered multicast, would suffice. After all, the bound is only a hint to help prune the search tree. Delayed, unordered, or undelivered new bounds will not influence the correctness of the solution, just the running time of the program. In the event of many bound updates, the performance improvements by relaxing the semantics for this single shared object may outweigh the need for sequential consistency.

Unlike the standard Orca system, in FlexRTS it is possible to add a shared object implementation that relaxes the sequential consistency requirement. By simply registering the new shared object implementation in the appropriate location, the run-time system will use it instead of its own built-in implementation. This special-purpose object implementation could use, for example, unreliable Ethernet multicasts to distribute updates. A different implementation could include the use of active filters, such as described in Section 4.3.2, which would only select messages that report a better bound than the one currently found by the local search process.

Successive Overrelaxation

Successive Overrelaxation (SOR) is a iterative method for solving discrete Laplace equations on a grid. SOR is a slight modification of the Gauss-Seidel algorithm that significantly improves the convergence speed. The SOR method computes the weighted average of its four neighbors between two iterations, *av*, and for each point in the grid it computes $M[r,c] = M[r,c] + \omega\,(av - M[r,c])$, where ω is the relaxation parameter (typically $0 < \omega < 2$). The algorithm terminates when the computation converges.

SOR is an inherently sequential process that can be parallelized by dividing the grid in subgrids such that each process is allocated a proportional share of consecutive grid columns [Bal, 1989]. Each process can then compute the SOR iterations but for the grid borders it has to communicate with the processes that hold the neighboring subgrid.

Admitted, SOR is a difficult program for Orca since it consists mainly of point-to-point communication between two neighbors rather than using group multicasts and,

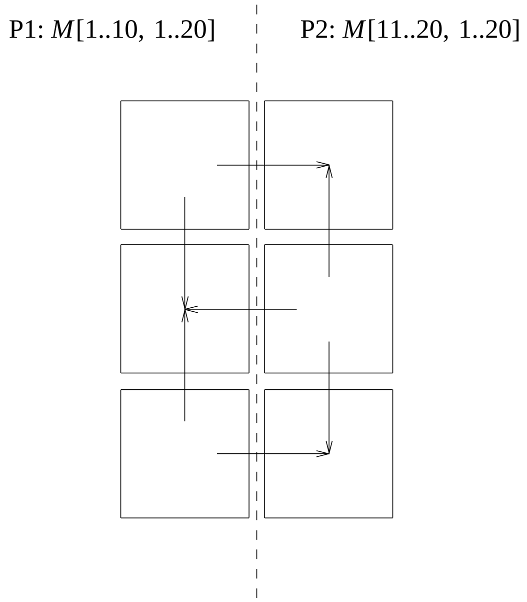


Figure 5.5. Boundary conditions for a parallel SOR algorithm.

as in the previous TSP example, guaranteeing sequential consistency is probably too strong since the algorithm converges toward a solution. Instead, a FIFO or PRAM ordering guarantee suffices. In this situation we could implement a special-purpose shared column object, representing a column shared between two neighbors, that uses a simple point-to-point communication protocol (*i.e.*, go-back-n or selective repeat) to update the shared state and batch updates. Using FlexRTS, the programmer can add this new shared object to the run-time system without changing the original Orca SOR program.

Iterative Deepening Algorithm

The iterative deepening A^* algorithm (IDA) is a provably optimal, in terms of memory usage and time complexity, heuristic tree search algorithm [Korf, 1985]. The IDA^* is a branch-and-bound type of algorithm based on the concept of depth-first iterative deepening but it prunes the search tree by using a heuristic. The heuristic is part of the A^* cost function which is used during the depth-first search to prune branches of the search tree. A typical application of the IDA^* algorithm is to solve a 15-puzzle. See Luger and Stubblefield [Luger and Stubblefield, 1989] for more detail on the IDA^* algorithm and its applications.

Our implementation of a parallel 15-puzzle solver consists of a number of parallel workers, a distributed job queue and the current search depth. Each worker process takes a job from the job queue and uses the IDA^* algorithm to search the game tree up to the specified search depth. If no solution is found at that depth, the left over subtrees are placed on the work queue. The workers continue until a solution is found or the job queue becomes empty.

This application can take advantage of FlexRTS by implementing special-purpose shared objects for the search depth and the job queue. The search depth value is controlled by the main process that determines the iteration depth and the variable is read by all worker processes. For this variable the shared integer object implementation described in the previous section is a good choice. The job queue, on the other hand, has a high read and write ratio and under the traditional run-time system would be implemented by a single copy. In FlexRTS we can implement a separate job queue object that would maintain a write back cache of jobs. That is, jobs accumulate at the process that generates it unless other processes run out or a certain threshhold is reached. Unlike the other examples, sequential consistency for the search depth and job queue shared objects is important in this application.

5.2. Secure Java Run Time System

Java™ [Gosling *et al.*, 1996] is a general-purpose programming language that has gained popularity as the programming language of choice for mobile computing where the computation is moved from the initiator to a client or server. The language is used for World Wide Web programming [Arnold and Gosling, 1997], smart card programming [Guthery and Jurgensen, 1998], embedded device programming [Esmertec, 1998], and even for providing executable content for active networks [Wetherall and Tennenhouse, 1996]. Three reasons for this popularity are Java's portability, its security properties, and its automatic memory allocation and deallocation.

Java programs are compiled into an intermediate representation called bytecodes and run on a Java Virtual Machine (JVM). This JVM contains a bytecode verifier that is essential for Java's security. Before execution begins the verifier checks that the byte codes do not interfere with the execution of other programs by assuring they use valid references and control transfers. Bytecodes that successfully pass this verification are executed but are still subject to a number of other security measures implemented in the Java run-time system.

All of Java's security mechanisms depend on the correct implementation of the bytecode verifier and a secure environment in which it can run. In our opinion this is a flawed assumption and past experience has shown a number of security problems with this approach [Dean *et al.*, 1996; Felten, 1999; Sirer, 1997]. More fundamental is that from software engineering research it is known that every 1000 lines of code contain 35-80 bugs [Boehm, 1981]. Even very thoroughly tested programs still contain on average about 0.5-3 bugs per 1000 lines of code [Myers, 1986]. Given that JDK 2 contains ~1.6M lines of code it is reasonable to expect 56K to 128K bugs. Granted, not all of these bugs are in security critical code, but all of the code is security sensitive since it runs within a single protection domain.

Other unsolved security problems with current JVM designs are its vulnerability to denial of service attacks and its discretionary access control mechanisms. Denial of service attacks are possible because the JVM lacks proper support to bound the amount of memory and CPU cycles used by an application. The discretionary access control model is not always the most appropriate one for executing untrusted mobile code on relatively insecure clients.

Interestingly, exactly the same security problems occur in operating systems. There they are solved by introducing hardware separation between different protection domains and controlled access between them. This hardware separation is provided by the memory management unit (MMU), an independent hardware component that controls all accesses to main memory. To control the resources used by a process an operating system limits the amount of memory it can use, assigns priorities to bias its scheduling, and enforces mandatory access control. However, unlike programming language elements, processes are coarse grained and have primitive sharing and communication mechanisms.

An obvious solution to Java's security problems is to integrate the JVM with the operating system's process protection mechanisms. How to adapt the JVM efficiently and transparently (*i.e.*, such that multiple Java applets can run on the same JVM while protected by the MMU) is a nontrivial problem. It requires a number of hard operating system problems to be resolved. These problems include: uniform object naming, object sharing, remote method invocation, thread migration, and protection domain and memory management.

The central goal of our work was the efficient integration of operating system protection mechanisms with a Java run-time system to provide stronger security guarantees. A subgoal was to be transparent with respect to Java programs. Where security and transparency conflicted they were resolved by a separate security policy. Using the techniques described in this paper, we have built a prototype JVM with the following features:

- Transparent, hardware-assisted separation of Java classes, provided that they do not violate a preset security policy.
- Control over memory and CPU resources used by a Java class.
- Enforcement of mandatory access control for Java method invocations, class inheritance, and system resources.
- Employment of the *least privilege* concept and the introduction of a *minimal trusted computing base* (TCB).
- The JVM does not depend on the correctness of the Java bytecode verifier for interdomain protection.

In our opinion, a JVM using these techniques is much more amenable to an ITSEC [UK ITSEC, 2000] or a Common Criteria [Common Criteria, 2000] evaluation than a pure software protection based system.

Our JVM consists of a small trusted component, called the *Java Nucleus*, which acts as a reference monitor and manages and mediates access between different protection domains (see Figure 5.6). These protection domains contain one of more Java classes and their object instances. The Java classes themselves are compiled to native machine code rather than being interpreted. The references to objects are capabilities [Dennis and Van Horn, 1966], which are managed by the Java Nucleus.

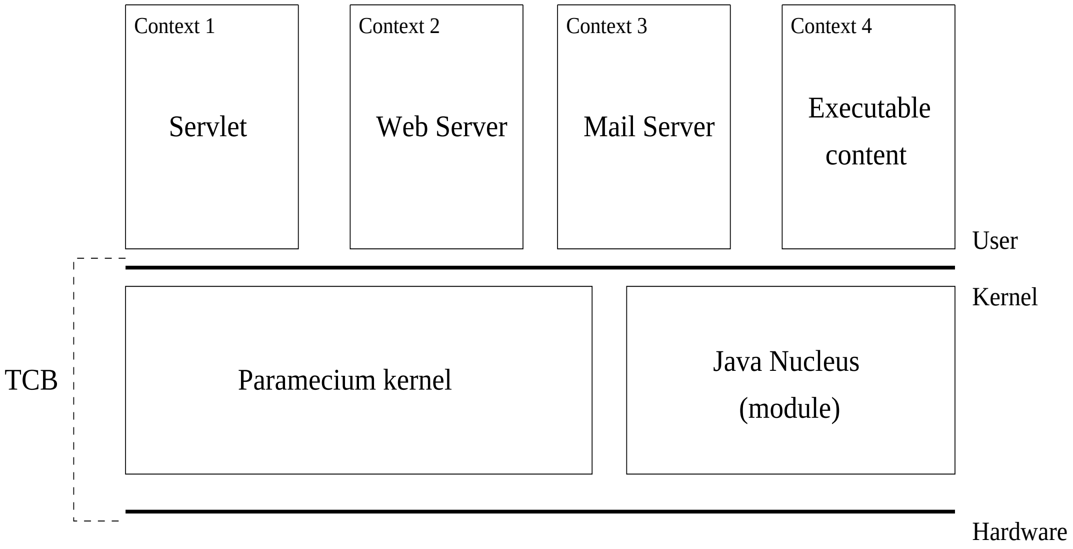


Figure 5.6. Secure JVM overview. In this example the Java Nucleus is instantiated as a kernel module and it mediates access between all the shown contexts.

For an efficient implementation of our JVM we depend on the low-level operating system functionality provided by *Paramecium* [Van Doorn *et al.*, 1995]. The Java Nucleus uses its low-level protection domain and memory management facilities for separation into protection domains and its IPC facility for cross domain method invocations. The data is shared on demand using virtual memory remapping. When the data contains pointers to other data elements, they are transparently shared as well. The garbage collector, which is a part of the Java Nucleus, handles run-time data relocation, sharing and revocation of data elements, protection, and the reclaiming of unused memory cells over multiple protection domains.

In the next section of this chapter we will describe the problems involved in the integration of a language and an operating system. Section 5.2.2 discusses the separation of concerns when designing a JVM architecture with a minimal TCB. It focuses on the security guarantees offered at run time and the corresponding threat model. Since our system relies on Paramecium primitives, we briefly repeat the features the system depends on in Section 5.2.3. Section 5.2.4 describes the key implementation details of our JVM. It discusses the memory model used by our JVM, its IPC mechanism, its data sharing techniques, and its garbage collector. Section 5.2.5 briefly discusses some implementation details and some early experiences with our JVM, including a performance analysis and some example applications. The related work, conclusions, and future extensions are described in Section 5.3.

5.2.1. Operating and Run Time System Integration

Integration of an operating system and a language run-time system has a long history (*e.g.*, Intel *i*APX 432 [Organick, 1983], Mesa/Cedar [Teitelman, 1984], Lisp Machines [Moon, 1991], Oberon [Wirth and Gütknecht, 1992], JavaOS [Saulpaugh and Mirho, 1999], *etc.*), but none of these systems used hardware protection to supplement the protection provided by the programming language. In fact, most of these systems provide no protection at all or depend on a trusted code generator. For example, the Burroughs B5000 [Burroughs, 1961] enforced protection through a trusted compiler. The Burroughs B5000 did not provide an assembler, other than one for in-house development of the diagnostic software, since it could be used to circumvent this protection.

Over the years these integrated systems have lost popularity in favor of time-shared systems with a process protection model. These newer systems provide better security and fault isolation by using hardware separation between untrusted processes and controlling the communication between them. A side effect of this separation is that sharing can be much harder and less efficient.

The primary reasons why the transparent integration of a process protection model and a programming language are difficult are summarized in Figure 5.7. The key problem is their lack of a common naming scheme. In a process model each process has its own virtual address space, requiring techniques like pointer swizzling to translate addresses between different domains. Aside from the naming issues, the sharing granularity is different. Processes can share coarse grained pages while programs share many small variables. Reconciling the two as in distributed shared memory systems [Li and Hudak, 1989] leads to the undesirable effects of false sharing or fragmentation. Another distinction is the unit of protection. For a process this is an protection domain, for programs it is a module, class, object, *etc*. Finally, processes use rudimentary IPC facilities to send and receive blocks of data. Programs, on the other hand, use procedure calls and memory references.

	Process Protection Model	**Programming Language**
Name space	Disjoint	Single
Granularity	Pages	Variables
Unit	Protection domain	Class/object
Communication	IPC	Call/memory

Figure 5.7. Process protection model *vs.* programming language.

In order to integrate a process protection model and a programming language we need to adapt some of the key process abstractions. Adapting them is hard to do in a traditional operating system because they are hardwired into the system. Extensible operating systems on the other hand provide much more flexibility (*e.g.,* Paramecium, OSKit [Ford *et al.*, 1997], L4/LavaOS [Liedtke *et al.*, 1997], ExOS [Engler *et al.*, 1995], and SPIN [Bershad *et al.*, 1995b]). For example, in our system the Java Nucleus acts as a special purpose kernel for Java programs. It controls the protection domains that contain Java programs, creates memory mappings, handles all protection faults for these domains, and controls cross protection domain invocations. These functions are hard to implement on a traditional system but straightforward on an extensible operating system. A second enabling feature of extensible operating systems is the dramatic improvement in cross domain transfer cost by eliminating unnecessary abstractions [Bershad *et al.*, 1989; Hsieh *et al.*, 1993; Liedtke *et al.*, 1997; Shapiro *et al.*, 1996]. This makes the tight integration of multiple protection domains feasible. Another advantage of using an extensible kernel is that they tend to be several orders of magnitude smaller than traditional kernels. This is a desirable property since the kernel is part of the TCB.

For a programming language to benefit from hardware separation it has to exhibit a number of requirements. The first one is that the language must contain a notion of a unit of protection. These units form the basis of the protection system. Examples of these units are classes, objects, and modules. Each of these units must have one or more interfaces to communicate with other units. Furthermore, there need to be non-language reasons to separate these units, like running multiple untrusted applets simultaneously on the same system. The last requirement is that the language needs to use a typed garbage collection system rather than programmer managed dynamic memory. This requirement allows a third party to manage, share and relocate the memory used by a program.

The requirements listed above apply to many different languages (*e.g.*, Modula3 [Nelson, 1991], Oberon [Wirth and Gütknecht, 1992], and ML [Milner *et al.*, 1990]) and operating systems (*e.g.*, ExOS [Engler *et al.*, 1994], SPIN [Bershad *et al.*, 1995b], and Eros [Shapiro *et al.*, 1999]), for our research we concentrated on the integration of Paramecium and Java. Before discussing the exact details of how we solved each integration difficulty, we first discuss the protection and threat model for our system.

5.2.2. Separation of Concerns

The goal of our secure JVM is to minimize the trusted computing base (TCB) for a Java run-time system. For this it is important to separate security concerns from language protection concerns and establish what type of security enforcement has to be done at compile time, load time, and run time.

At compile time the language syntax and semantic rules are enforced by a compiler. This enforcement ensures valid input for the transformation process of source code into bytecodes. Since the compiler is not trusted, the resulting bytecodes cannot be trusted and, therefore, we cannot depend on the compiler for security enforcement.

At load time a traditional JVM loads the bytecodes and relies on the bytecode verifier and various run-time checks to enforce the Java security guarantees. As we discussed in the introduction to this section, we do not rely on the Java bytecode verifier for security based on its size, complexity, and poor track record. Instead, we aim at minimizing the TCB, and use hardware fault isolation between groups of classes and their object instances, and control access to methods and state shared between them. A separate security policy defines which classes are grouped together in a single protection domain and which methods they may invoke on different protection domains. It is important to realize that all classes within a single protection domain have the same trust level. Our system provides strong protection guarantees between different protection domains, *i.e.*, interdomain protection. It does not enforce intradomain protection; this is left to the run-time system if desirable. This does not constitute a breakdown of security of the system. It is the policy that defines the security. If two classes in the same domain, *i.e.*, have the same trust level, misbehave with respect to one another, this clearly constitutes a failure in the policy specification. These two classes should not have been in the same protection domain.

The run-time security provided by our JVM consists of hardware fault isolation among groups of classes and their object instances by isolating them into multiple protection domains and controlling access to methods and state shared between them. Each security policy, a collection of permissions and accessible system resources, defines a protection domain. All classes with the same security policy are grouped into the same domain and have unrestricted access to the methods and state within it. Invocations of methods in other domains pass through the Java Nucleus. The Java Nucleus is a trusted component of the system and enforces access control based on the security policy associated with the source and target domain.

From Paramecium’s point of view, the Java Nucleus is a module that is loaded either into the kernel or into a separate user context. Internally, the Java Nucleus consists of four components: a class loader, a garbage collector, a thread system, and an IPC component. The class loader loads a new class, translates the bytecodes into native machine codes, and deposits them into a specified protection domain. The garbage collector allocates and collects memory over multiple protection domains, assists in sharing memory among them, and implements memory resource control. The thread system provides the Java threads of control and maps them directly onto Paramecium threads. The IPC component implements cross protection domain invocations, access control, and CPU resource usage control.

The JVM trust model (*i.e.*, what is included in the minimal trusted computing base) depends on the correct functioning of the garbage collector, IPC component, and thread system. We do not depend on the correctness of the bytecode translator. However, if we opt to put some minimal trust in the byte code translator, it enables certain optimizations that are discussed below.

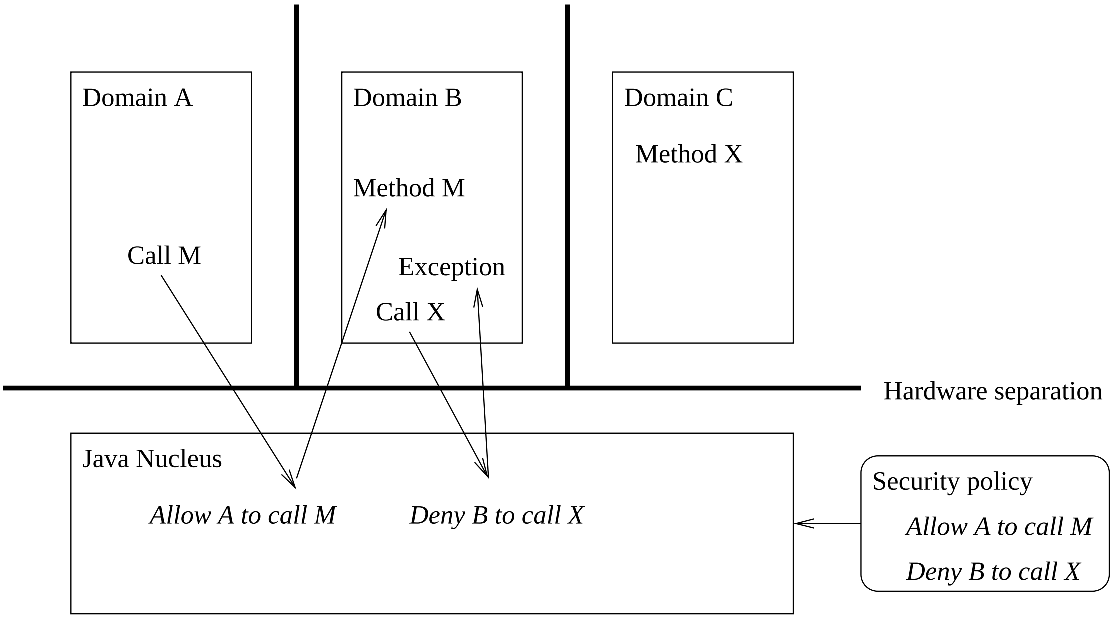


Figure 5.8. The Java Nucleus uses hardware protection to separate Java classes, which are placed in separate protection domains The Java Nucleus uses a security policy to determine which domain can call which methods and enforce access control.

References to memory cells (primitive types or objects) act as capabilities [Dennis and Van Horn, 1966] and can be passed to other protection domains as part of a cross domain method invocation (XMI) or object instance state sharing. Passing an object reference results in passing the full closure of the reference. That is, all cells that can be obtained by dereferencing the pointers that are contained in the cell of which the reference is passed, without this resulting in copying large amounts of data, a so-called deep copy. Capabilities can be used to implement the notion of least privilege but capabilities also suffer from the classical confinement and revocation problem [Boebert, 1984; Karger and Herbert, 1984]. Solving these is straightforward since the Java Nucleus acts as a reference monitor. However, this violates the Java language transparency requirement (see Section 5.3).

Our system does not depend on the Java security features such as bytecode verification, discretionary access control through the security manager, or its type system. We view these as language security measures that assist the programmer during program development which should not be confused or combined with system security measures. The latter isolates and mediates access between protection domains and resources; these measures are independent of the language. However, integrating operating system style protection with the semantic information provided by the language run-time system does allow finer grained protection and sharing than is possible in contemporary systems.

Granularity	**Mechanism**
Method	Invocation access control
Class	Instruction text sharing between domains
Class	Object sharing between domains
Reference	Opaque object handle
System	Paramecium name space per domain

Figure 5.9. Security policy elements.

The security provided by our JVM is defined in a security policy. The elements that comprise this policy are listed in Figure 5.9. They consist of a set of system resources available to each protection domain, classes whose implementation is shared between multiple domains, object instance state that is shared, and access control for each cross domain method invocation.

The first policy element is a per method access control for cross protection domain invocations. Each method has associated with it a list of domains that can invoke it. A *domain* is similar to, and in fact implemented as, a Paramecium context, but unlike a context, it is managed by the Java Nucleus which controls all the mappings and exceptions for it. If the invocation target is not in this domain list, access is denied. Protection is between domains, not within domains, hence there is no access control for method invocations within the same domain.

To reduce the amount of memory used and the number of cross protection domain calls (XMIs), the class text (instructions) can be shared between multiple domains. This is analogous to text sharing in UNIX, where the instructions are loaded into memory only once and mapped into each domain that uses it in order to reduce memory requirements. In our case it eliminates the need for expensive XMIs. The object instance state is still private to each domain.

Object instance state is transparently shared between domains when references to it are passed over XMIs or when an object inherits from a class in a different protection domain. Which objects can be passed between domains is controlled by the Java programs and not by the JVM. Specifying this as part of the security policy would break the Java language transparency requirement. Per-method access control gives the JVM the opportunity to indirectly control which references are passed.

In circumstances where object instance state sharing is not desirable, a class can be marked as nonsharable for a specified domain. Object references of this class can still be passed to the domain but cannot be dereferenced by it. This situation is similar to client/server mechanisms where the reference acts as an opaque object handle. Since

Java is not a pure object-oriented language (*e.g.*, it allows clients to directly access object state) this mechanism is not transparent for some Java programs.

Fine grained access control of the system resources is provided by the Paramecium name space mechanism. If a service name is not in the name space of a protection domain, that domain cannot gain access to the service. The name space for each protection domain is constructed and controlled by our Java Nucleus.

To further reduce the number of XMIs, classes with the same security policy are grouped into the same protection domain. The number of XMIs can be reduced further still by sharing the instruction text of class implementations between different domains. The need for minimizing the numbers of XMIs was underscored by Colwell's thesis [Colwell, 1985] which discussed the of performance problems with the Intel *i*APX 432.

Threat	**Protection mechanism**
Fault isolation	Protection domains
Denial of service	Resource control
Forged object references	Garbage collector
Illegal object invocations	XMI access control

Figure 5.10. Threat model.

Figure 5.10 summarizes the potential threats our JVM can handle, together with their primary protection mechanism. Some threats, such as covert channels, are not handled in our system. Other threats, such as denial of service attacks caused by improper locking behavior are considered policy errors. The offending applet should not have been given access to the mutex.

5.2.3. Paramecium Integration

Our secure Java virtual machine design relies heavily on the availability of low-level system services such as efficient IPC, memory management and name space management. In this section we discuss how the Paramecium kernel enables our JVM and we define the functionality upon which the Java Nucleus depends.

The key to our JVM design is the ability to efficiently communicate between different protection domains. For this we utilize Paramecium's event and invocation chain mechanisms. Our event mechanism provides an efficient cross protection domain communication mechanism by raising an event that causes a handler to be invoked instantaneously in a different domain. Part of this invocation is that a predetermined number of arguments are passed from the invoker to the handler. A sequence of event invocations caused by a single thread of control is called an invocation chain and forms the basis for our migrating thread package. This thread package efficiently manages multiple threads operating in different protection domains.

The second enabling service provided by our kernel is memory and protection domain management, of which memory management is divided into physical and virtual memory management. These separate services are used extensively by the Java Nucleus and allows it to have fine grained control over the organization of the virtual memory address spaces. The Java Nucleus uses it to create and manage a single shared address space among multiple protection domains with each protection domain containing one or more Java classes. The exact memory mapping details are described in the next section, but enabling the Java Nucleus to control the memory mappings on a per page basis for these protection domains and handle all their fault events is crucial for the design of the JVM. This, together with name space management, enables the Java Nucleus to completely control the execution environment for the protection domains and, consequently, the Java programs it manages.

Each protection domain has a private name space associated with it. It is hierarchical and contains all the object instance names which the protection domain can access. The name space is populated by the parent and it determines which object instances its children can access. In the JVM case, the Java Nucleus creates all the protection domains it manages and consequently it populates the name spaces for these domains. For most protection domains this name space is empty and the programs running in them can only communicate with the Java Nucleus using its cross domain method invocation (XMI) mechanism. These programs cannot communicate directly with the kernel or any other protection domain in the system since they do not have access to the appropriate object instances. Neither can they fabricate them since the names and proxies are managed by the kernel. Under certain circumstances determined by the security policy, a protection domain can have direct access to a Paramecium object instance and the Java Nucleus will populate the protection domain's name space with it. This is mostly used to allow Java packages, such as the windowing toolkits, to efficiently access the system resources.

The Java Nucleus is a separate module that is either instantiated in its own protection domain or as an extension colocated with the kernel. Colocating the Java Nucleus with the kernel does not reduce the security of the system since both are considered part of the TCB. However, it does improve the performance since it reduces the number of cross protection domain calls required for communicating with the managed protection domains.

5.2.4. Secure Java Virtual Machine

The Java Nucleus forms the minimal trusted computing base (TCB) of our secure JVM. This section describes the key techniques and algorithms used by the Java Nucleus.

In short, the Java Nucleus provides a uniform naming scheme for all protection domains, including the Java Nucleus. It provides a single virtual address space where each protection domain can have a different protection view. All cross protection domain method invocations (XMIs) pass through our Java Nucleus, which controls access, CPU, and memory resources. Data is shared on demand between multiple protection domains, that is, whenever a reference to shared data is dereferenced. Our Java Nucleus uses shared memory and runtime reallocation techniques to accomplish this. Only references passed over an XMI or object instances whose inherited classes are in different protection domains can be accessed; others will cause security violations. These protection mechanisms depend on our garbage collector to allocate and deallocate typed memory, relocate memory, control memory usage, and keep track of ownership and sharing status.

The Java Nucleus uses on-the-fly compilation techniques to compile Java classes into native machine code. It is possible to use the techniques described below to build a secure JVM using an interpreter rather than a compiler. Each protection domain would then have a shared copy of the interpreter interpreting the Java bytecodes for that protection domain. We have not explored such an implementation because of the obvious performance loss of interpreting bytecodes.

The next subsections describe the key techniques and algorithms in greater detail.

Memory Organization

The Java Virtual Machine model assumes a single address space in which multiple applications can pass object references to each other by using method invocations and shared memory areas. This, and Java's dependence on garbage collection, dictated our memory organization.

Inspired by single address space operating systems [Chase *et al.*, 1994], we have organized memory into a single virtual address space. Multiple, possibly unrelated, programs live in this single address space. Each protection domain has, depending on its privileges, a view onto this address space. This view includes a set of virtual to physical memory page mappings together with their corresponding access rights. A small portion of the virtual address space is reserved by each protection domain to store domain specific data.

Central to the protection domain scheme is the Java Nucleus (see Figure 5.11). The Java Nucleus is analogous to an operating system kernel. It manages a number of protection domains and has full access to all memory mapped into these domains and their corresponding access permissions. The protection domains themselves cannot manipulate the memory mappings or the access rights of their virtual memory pages. The Java Nucleus handles both data and instruction access (*i.e.*, page) faults for these domains. Page faults are turned into appropriate Java exceptions when they are not handled by the system.

For convenience all the memory available to all protection domains is mapped into the Java Nucleus with read/write permission. This allows it to quickly access the data in different protection domains. Because memory addresses are unique and the memory pages are mapped into the Java Nucleus protection domain, the Java Nucleus does not have to map or copy memory as an ordinary operating system kernel.

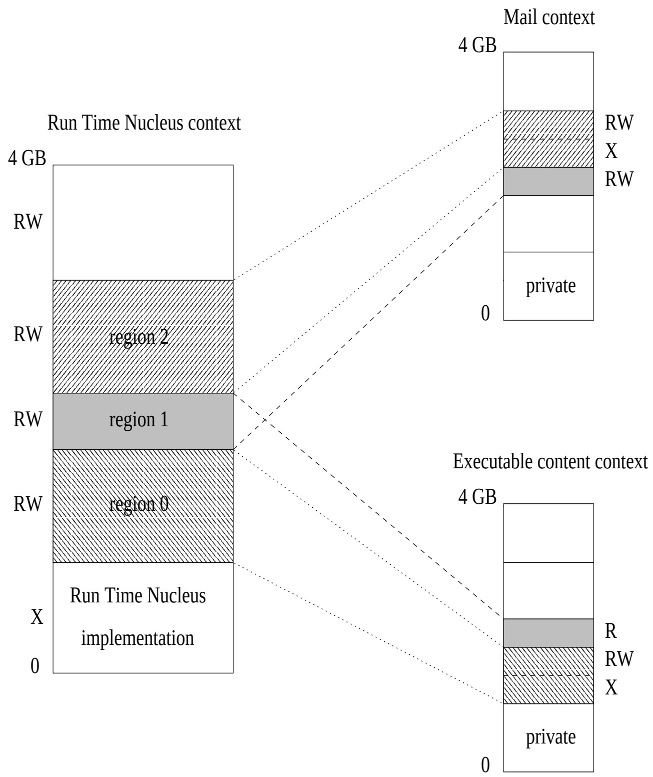


Figure 5.11. Java Nucleus virtual memory map. The Java nucleus has full read/write access to all memory used by the application contexts. The application contexts have the same view but with different page protection attributes.

The view different protection domains have of the address space depends on the mappings created by the Java Nucleus. Consider Figure 5.11. A mail reader application resides in the context named mail. For efficiency reasons, all classes constituting this application reside in the same protection domain; all executable content embedded in an e-mail message is executed in a separate domain, say executable content. In this example memory region 0 is mapped into the context executable content. Part of this memory contains executable code and has the execute privilege associated with it. Another part contains the stack and data and has the read/write privilege. Region 0 is only visible to the executable content context and not to the mail context. Likewise, region 2 is not visible to the executable content context. Because of the hardware memory mappings these two contexts are physically separated.

Region 1 is used to transfer data between the two contexts and is set up by the Java Nucleus. Both contexts have access to the data, although the executable content context has only read access. Violating this access privilege causes a data access fault to be generated which is handled by the Java Nucleus. It will turn the fault into a Java exception.

Cross Domain Method Invocations

A cross domain method invocation (XMI) mimics a local method invocation except that it crosses a protection domain boundary. A vast amount of literature exists on low latency cross domain control transfer [Bershad *et al.*, 1989; Hsieh *et al.*, 1993; Liedtke *et al.*, 1997]. Our XMI mechanism is loosely based on Paramecium's system call mechanism, which uses events. The following example illustrates the steps involved in an XMI.

Consider the protection domains A and B and a method M which resides in domain B and a thread executing in domain A which calls method M. The Java Nucleus generated the code in domain A and filled in the real virtual address for method M. Hence, domain A knows the address for function M, but it does not have access to the pages which contain the code for function M. These are only mapped into domain B. Hence, when A calls method M an instruction fault will occur since the code for M is not mapped into context A. The fault causes an event to be raised in the Java Nucleus. The event handler for this fault is passed two arguments: the fault address (*i.e.*, method address) and the fault location (*i.e.*, call instruction). Using the method address, the Java Nucleus determines the method information which contains the destination domain and the access control information. Paramecium's event interface is used to determine the caller domain. Based on this information, an access decision is made. If access is denied, a security exception is raised in the caller domain.

Using the fact that method information is static and that domain information is static for code that is not shared, we can improve the access control check process. Rather than looking up this information, the Java Nucleus stores a pointer to it in the native code segment of the calling domain. The information can then be accessed quickly using a fixed offset and fault location parameter. Method calls are achieved through special trampoline code that embeds these two values. More precisely, the call trampoline code fragment in context A for calling method M appears as (in SPARC [Sun Microsystems Inc., 1992] assembly):

```
    call  M               ! call method M
      mov   %g0, %i0      ! nil object argument
    b,a   next_instr      ! branch over
    .long <caller domain> !   JNucleus caller domain pointer
    .long <method info>   !   JNucleus method info pointer
next_instr:
```

The information stored in the caller domain must be protected from tampering. This is achieved by mapping all executable native code as execute only; only the Java Nucleus has full access to it.

When access is granted for an XMI, an event is associated with the method if one is not already present. Then the arguments are copied into the registers and onto the event handler stack as dictated by the calling frame convention. No additional marshaling of the parameters is required. Both value and reference parameters are passed unchanged. Using the method's type signature to identify reference parameters, we mark data references as exported roots (*i.e.*, garbage collection roots). Instance data is mapped on demand as described in the next section. Invoking a method on an object reference causes an XMI to the method implementation in the object owner's protection domain.

Virtual method invocations, where a set of specific targets is known at compile-time but the actual target only at runtime, require a lookup in a switch table. The destinations in this table refer to call trampolines rather than the actual method address. Each call trampoline consists of the code fragment described above.

Using migratory threads, an XMI extends the invocation chain of the executing thread into another protection domain. Before raising the event to invoke the method, the Java Nucleus adjusts the thread priority according to the priority of the destination protection domain. The original thread priority is restored on the method return. Setting the thread priority enables the Java Nucleus to control the CPU resources used by the destination protection domain.

The Java Nucleus requires an explicit event invoke to call the method rather than causing an instruction fault which is handled by the destination domain. The reason for this is that it is not possible in Paramecium to associate a specific stack (*i.e.*, the one holding the arguments) with a fault event handler. Hence the event has to be invoked directly. This influences the performance of the system depending on whether the Java Nucleus is instantiated in a separate address space or as a module in the kernel. When the Java Nucleus is in a separate process, an extra system call is necessary to enter the kernel. The invoked routine is called directly when the Java Nucleus is instantiated as a kernel module.

Local method invocations use the same method call trampoline as the one outlined above, except that the Java Nucleus does not intervene. This is because the method address is available locally and does not generate a fault. The uniform trampoline allows the Java Nucleus to share class implementations among multiple protection domains by mapping them in. For example, simple classes like the java.lang.String or java.lang.Long can be shared by all protection domains without security implications. Sharing class implementations reduces memory use and improves performance by eliminating XMIs. XMIs made from a shared class do not have their caller domain set, since there can be many caller domains, and require the Java Nucleus to use the system authentication interface to determine the caller.

Data Sharing

Passing parameters, as part of a cross domain method invocation (XMI), requires simply copying them by value and marking the reference variables as exported roots. That is, the parameters are copied by value from the caller to the callee stack without dereferencing any of the references. Subsequent accesses to these references will cause a protection fault unless the reference is already mapped in. The Java Nucleus, which handles the access fault, will determine whether the faulting domain is allowed access to the variable referenced. If allowed, it will share the page on which the variable is located.

Sharing memory on a page basis traditionally leads to false sharing or fragmentation. Both are clearly undesirable. False sharing occurs when a variable on a page is mapped into two address spaces and the same page contains other unrelated variables. This clearly violates the confinement guarantee of the protection domain. Allocating each variable on a separate page results in fragmentation with large amounts of unused physical memory. To share data efficiently between different address spaces, we use the garbage collector to reallocate the data at runtime. This prevents false sharing and fragmentation.

Consider Figure 5.12 which shows the remapping process to share a variable *a* between the mail context and the executable content context. In order to relocate this variable we use the garbage collector to update all the references. To prevent race conditions the threads within or entering the contexts that hold a reference to *a* are suspended (step 1). Then the data, *a*, is copied onto a new memory page (or pages depending on its size) and referred to as a'. The other data on the page is not copied, so there is no risk of false sharing. The garbage collector is then used to update all references that point to *a* into references that point to a' (step 2). The page holding a' is then mapped into the other context (step 3) Finally, the threads are resumed, and new threads are allowed to enter the unblocked protection domains (step 4). The garbage collector will eventually delete *a* since it does not have any references to it.

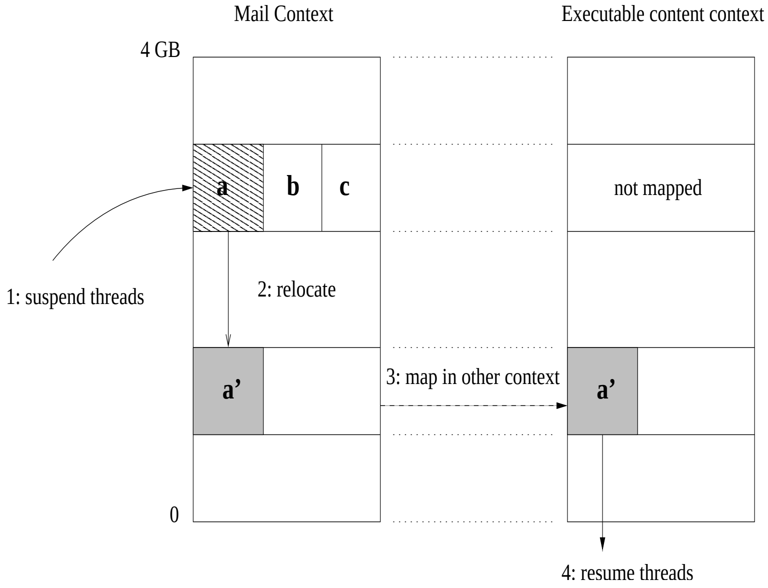


Figure 5.12. Data remapping between different protection domains. All protection domains share the same address space but have different mapping and protection views.

Other variables that are shared between the same protection domains are tagged onto the already shared pages to reduce memory fragmentation. The process outlined above can be applied recursively. That is, when a third protection domain needs access to a shared variable the variable is reallocated on a page that is shared between the three domains.

In order for the garbage collector (see below) to update the cell references it has to be exact. That is, it must keep track of the cell types and of references to each cell to distinguish valid pointers from random integer values. The updating itself can either be done by a full walk over all the in-use memory cells or by arranging each cell to keep track of the objects that reference it. The overhead of the relocation is amortized over subsequent uses.

Besides remapping dynamic memory, the mechanism can also be used to remap static (or class) data. Absolute data memory references can occur within the native code generated by the just-in-time compiler. Rather than updating the data locations embedded in the native code on each data relocation, the just-in-time compiler generates an extra indirection to a placeholder holding the actual reference. This placeholder is registered with the garbage collector as a reference location.

Data remapping is used not only to share references passed as parameters over an XMI, but also to share object instance data between sub and superclasses in different protection domains. Normally, object instances reside in the protection domain in which their class was loaded. Method invocations on that object from different protection domains are turned into XMIs. In the case of an extended (*i.e.*, inherited) class the object instance state is shared between the two protection domains. This allows the sub and superclass methods to directly access the instance state rather than capturing all these accesses and turning them into XMIs. To accomplish this our JVM uses the memory remapping technique outlined above.

The decision to share object instance state is made at the construction time of the object. Construction involves calling the constructor for the class followed by the constructors for its parent classes. When the parent class is in a different protection domain the constructor invocation is turned into an XMI. The Java Nucleus performs the normal access control checks as for any other XMI from a different protection domain. The object instance state, that is passed implicitly as the first argument to the constructor call, is marked as an exportable root. The mechanisms involved in marking memory as an exportable root are discussed in the next section.

Java uses visibility rules (*i.e.*, *public* and *protected*) to control access to parts of the object instance state. Enforcing these rules through memory protection is straightforward. Each object's instance state is partitioned into a shared and nonshared part. Only the shared state can be mapped.

An example of state sharing between super and subclass is shown in Figure 5.13. Here the class BitMap and all its instances reside in protection domain A. Protection domain B contains all the instances of the class Draw. This class is an extension of the

```
class BitMap { // Domain A
    private static int N = 8, M = 8;
    protected byte bitmap[][];

    protected BitMap() {
        bitmap = new byte[N/8][M];
    }

    protected void set(int x, int y) {
        bitmap[x/8][y] |= 1<<(x%8);
    }
}

class Draw extends BitMap { // Domain B
    public void point(int x, int y) {
        super.set(x, y);
    }

    public void box(int x1, int y1,
                    int x2, int y2) {
        for (int x = x1; x < x2; x++)
            for (int y = y1; y < y2; y++)
                bitmap[x/8][y] |= 1<<(x%8);
    }
}
```

Figure 5.13. Simple box drawing class.

BitMap class which resides in a different protection domain. When a new instance of Draw is created the Draw constructor is called to initialize the class. In this case the constructor for Draw is empty and the constructor for the superclass BitMap is invoked. Invoking this constructor will cause a transfer into the Java Nucleus.

The Java Nucleus first checks the access permission for domain B to invoke the BitMap constructor in domain A. If granted, the object pointer is marked as an exportable root and passed as the first implicit parameter. Possible other arguments are copied as part of the XMI mechanism and the remote invocation is performed (see Figure 5.14). The BitMap constructor then assigns a new array to the bitmap field in the Draw object. Since the assignment is the first dereference for the object it will be remapped into domain A (see Figure 5.15).

When the creator of the Draw object calls box, see Figure 5.16, and dereferences bitmap it will be remapped into domain B (because the array is reachable from an exported root cell to domain A; see next section). Further calls to box do not require this remapping. A call to point results in an XMI to domain A where the superclass implementation resides. Since the Draw object was already remapped by the constructor it is does not require any remapping.

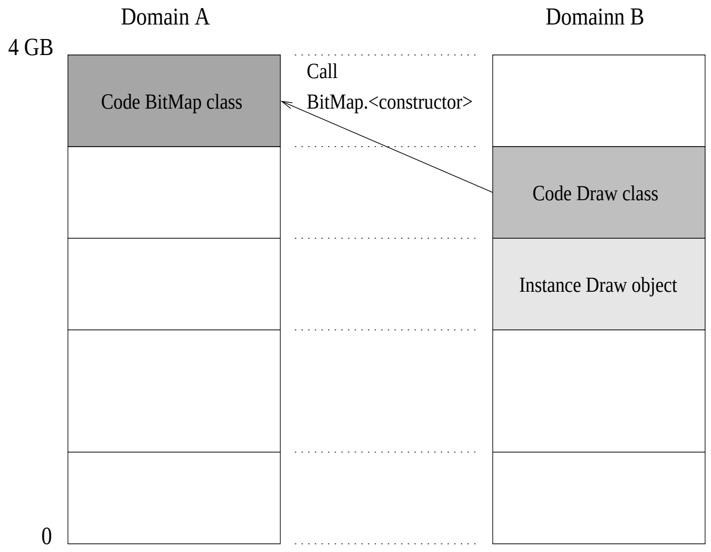


Figure 5.14. An instance of Draw has been created and the constructor for the super class BitMap is called. At this point the object instance state is only available to domain B.

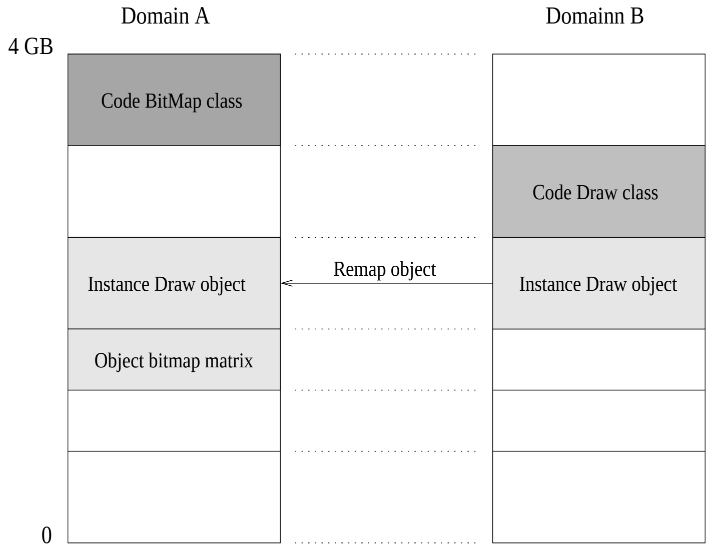


Figure 5.15. The BitMap constructor dereferences the object pointer, which causes the object to be mapped into domain B. At this point the object instance state is shared between domain A and B. The bitmap matrix that is allocated by the constructor is still local to domain A.

Whenever a reference is shared among address spaces, all references that are reachable from it are also shared and will be mapped on demand when referred to.

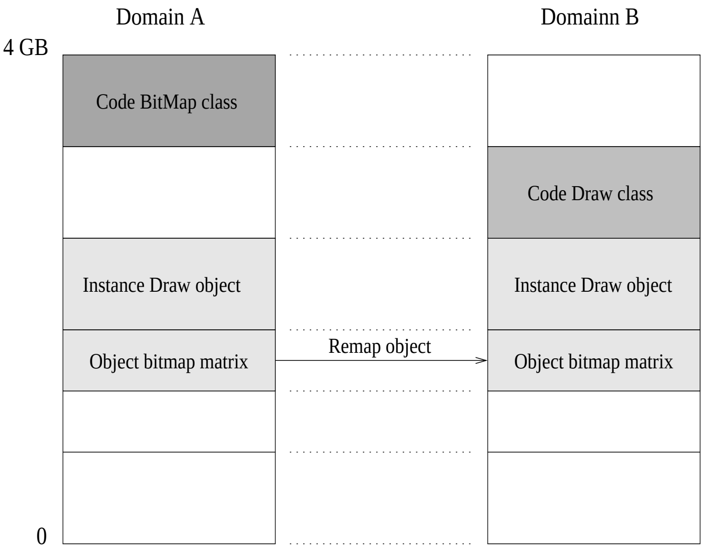


Figure 5.16. The Draw.box method is invoked and it dereferences the bitmap matrix. This will cause the bitmap matrix to be mapped into domain B.

This provides full transparency for Java programs which assume that a reference can be passed among all its classes. A potential problem with on-demand remapping is that it dilutes the programmers' notion of what is being shared over the life-time of a reference. This might obscure the security of the system. To strengthen the security, an implementation might decide not to support remapping of objects at all or provide a proactive form of instance state sharing. Not supporting instance state sharing prevents programs that use the object oriented programming model from being separated into multiple protection domains. For example, it precludes the isolation and sharing of the AWT package in a separate protection domain.

The implementation has to be conservative with respect to references passed as arguments to cross domain method invocations and has to unmap them whenever possible to restrict their shared access. Rather than unmapping at the invocation return time, which would incur a high call overhead, we defer this until garbage collection time. The garbage collector is aware of shared pages and determines whether they are reachable in the context they are mapped in. If they are unreachable, rather than removing all the bookkeeping information the page is marked invalid so it can be remapped quickly when it is used again.

Garbage Collection

Java uses garbage collection [Jones and Lins, 1996] to reclaim unused dynamic memory. Garbage collection, also known as automatic storage reclamation, is a technique whereby the run-time system determines whether memory is no longer used and can be reclaimed for new allocations. The main advantage of garbage collection is that it simplifies storage allocation for the programmer and makes certain programming errors, such as dangling pointers, impossible. The disadvantage is typically the increased memory usage and, to a lesser extent, the cost of performing the actual garbage collection.

A number of different garbage collection techniques exist. The two systems we refer to in this section are traced garbage collectors and conservative garbage collectors. A *traced garbage collector* organizes memory into many different *cells* which are the basic unit of dynamic storage. These cells are allocated and contain data structures ranging from single integers, strings, arrays to complete records. The garbage collector system knows about the layout of a cell and more specifically it knows where the pointers to other cells are. Each time the collector needs to find unused memory it will start from a set of cells, called the *root set,* and traverses each pointer in a cell to other cells and marks the ones it has seen. Eventually, the collector will find no unmarked cells that are reachable from the given root set. At that point all the marked cells are still in use and the unmarked ones are free to be reclaimed. This collection process gets more complicated, as described below, when you take into account that the *mutators*, the threads allocating memory and modifying it, run concurrently with the garbage collector and that the collector may run incrementally.

A *conservative garbage collector* uses a very different garbage collection process. It too allocates memory in the form of cells and keeps a list of allocated cells. At garbage collection time it traverses the content of each cell and any value in it that corresponds to a valid cell address is taken to be a reference to that cell and is therefore marked. This may well includes random values and the collector might mark cells as in-use while they are in fact not used. The advantage of this algorithm is that it does not require the garbage collection system to understand the layout of a cell.

Garbage collection is an integral part of the Java language and for our design we used a noncompacting incremental traced garbage collector that is part of the Java Nucleus. Our garbage collector is responsible for collecting memory in all the address spaces the Java Nucleus manages. A centralized garbage collector has the advantage that it is easier to share memory between different protection domains and to enforce central access and resource control. An incremental garbage collector has better real time properties than non-incremental collectors.

More precisely, the garbage collector for our secure Java machine must have the following properties:

1. Collect memory over multiple protection domains and protect the bookkeeping information from the potentially hostile domains.

2. Relocate data items at runtime. This property is necessary for sharing data across protection domains. Hence, we use an exact garbage collector rather than a conservative collector [Boehm and Weiser, 1988].

3. Determine whether a reference is reachable from an exported root. Only those variables that can be obtained via a reference passed as an XMI argument or instance state are shared.

4. Maintain, per protection domain, multiple memory pools with different access attributes. These are execute only, read-only, and read-write pools that contain native code, read-only and read-write data segments respectively.
5. Enforce resource control limitations per protection domain.

As discussed in the previous section all protection domains share the same virtual address map albeit with different protection views of it. The Java Nucleus protection domain, which contains the garbage collector, has full read-write access to all available memory. Hence the ability to collect memory over different domains is confined to the Java Nucleus.

A key feature of our garbage collector is that it integrates collection and protection. Classical tracing garbage collection algorithms assume a single address space in which all memory cells have the same access rights. A *cell* is a typed unit of storage which may be as small as an integer or contain more complex data structure definitions. For example, a cell may contain pointers for other cells. In our system cells have different access rights depending on the protection domain accessing it and cells can be shared among multiple domains. Although access control is enforced through the memory protection hardware, it is the garbage collector that has to create and destroy the memory mappings.

The algorithm we use (see the pseudo-code in Figure 5.17) is an extension of a classic mark-sweep algorithm which runs concurrently with the *mutators* (the threads modifying the data) [Dijkstra *et al.*, 1978]. The original algorithm uses a tricolor abstraction in which all cells are painted with one of the following colors: *black* indicates that the cell and its immediate descendents have been visited and are in use; *grey* indicates that the cell has been visited but not all of its descendents, or that its connectivity to the graph has changed; and *white* indicates untraceable (*i.e.*, free) cells. The garbage collection phase starts with all cells colored white and terminates when all traceable cells have been painted black. The remaining white cells are free and can be reclaimed.

To extend this algorithm to multiple protection domains we associate with each cell its owner domain and an export set. An export set denotes to which domains the cell has been properly exported. Garbage collection is performed on one protection domain at a time, each keeping its own color status to assist the marking phase. The marking phase starts by coloring all the root and exported root cells for that domain as grey. It then continues to examine all cells within that domain. If one of them is grey it is painted black and all its children are marked grey until there are no grey cells left. After the marking phase, all cells that are used by that domain are painted black. The virtual pages belonging to all the unused white cells are unmapped for that domain. When the cell is no longer used in any domain it is marked free and its storage space is reclaimed. Note that the algorithm in Figure 5.17 is a simplification of the actual implementation, many improvements (such as [Doligez and Gonthier, 1994; Kung and

```
COLLECT():
    for (;;) {
      for (d in Domains)
        MARK(d)
      SWEEP();
    }

MARK(d: Domain): // marker phase
    color[d, (exported) root set] = grey
    do {
      dirty = false
      for (c in Cells) {
        if (color[d, c] == grey) {
          color[d, c] = black
          for (h in children[c]) {
            color[d,h] = grey
            if (EXPORTABLE(c, h))
              export[d,h] |= export[d,c]
          }
          dirty = true
        }
      }
    } while (dirty)

SWEEP(): // sweeper phase
    for (c in Cells) {
      used = false
      for (d in Domains) {
        if (color[d, c] == white) {
          export[d, c] = nil
          UNMAP(d, c)
        } else
          used = true
        color[d, c] = white
      }
      if (used == false)
        DELETE(c)
    }

ASSIGN(a, c): // pointer assignment
    *a = c
    d = current domain
    export[d,c] |= export[d,a]
    if (color[d, c] == white)
      color[d, c] = grey

EXPORT(d: Domain, c: Cell): // export object
    color[d,c] = grey
    export[d,c] |= owner(c)
    export[owner(c),c] |= d
```

Figure 5.17. Multiple protection domain garbage collection.

Song, 1977; Steele, 1975]) are possible. A correctness proof of the algorithm follows from Dijkstra's paper [Dijkstra *et al.*, 1978].

Cells are shared between other protection domains by using the remapping technique described in the previous section. In order to determine whether a protection domain *d* has access to a cell *c*, the Java Nucleus has to examine the following three cases: The trivial case is where the owner of *c* is *d*. In this case the cell is already mapped into domain *d*. In the second case the owner of *c* has explicitly given access to *d* as part of an XMI parameter or instance state sharing or is directly reachable from such an exported root. This is reflected in the export information kept by the owner of *c*. Domain *d* has also access to cell *c* if there exists a transitive closure from some exported root *r* owned by the owner of *c* to some domain *z*. From this domain *z* there must exist an explicit assignment which resulted in *c* being inserted into a data structure owned by *d* or an XMI from the domains *z* to *d* passing cell *c* as an argument. In the case of an assignment the data structure is reachable from some other previously exported root passed by *d* to *z*. To maintain this export relationship each protection domain maintains a private copy of the cell export set. This set, usually nil and only needed for shared memory cells, reflects the protection domain's view of who can access the cell. A cell's export set is updated on each XMI (*i.e.*, export) or assignment as shown in Figure 5.17.

Some data structures, for example, exist prior to an XMI passing a reference to it. The export set information for these data structures is updated by the marker phase of the garbage collector. It advances the export set information from a parent to all its siblings taking the previously mentioned export constraints into account.

Maintaining an export set per domain is necessary to prevent forgeries. Consider a simpler design in which the marker phase advances the export set information to all siblings of a cell. This allows the following attack where an adversary forges a reference to an object in domain *d* and then invokes an XMI to *d* passing one of its data structures which embeds the forged pointer. The marker phase would then eventually mark the cell pointed to by the forged reference as exported to *d*. By maintaining for each cell a per protection domain export set forged pointers are impossible.

Another reason for keeping a per protection domain export set is to reduce the cost of a pointer assignment operation. Storing the export set in the Java Nucleus would require an expensive cross protection domain call for each pointer assignment, by keeping it in user space this can be eliminated. Besides, the export set is not the only field that needs to be updated. In the classic Dijkstra algorithm the cell's color information needs to be changed to *grey* on an assignment (see Figure 5.17). Both these fields are therefore kept in user space.

The cell bookkeeping information consists of three parts (see Figure 5.18). The public part contains the cell contents and its per domain color and export information. These parts are mapped into the user address space, where the color and export information is stored in the per domain private memory segment (see above). The nucleus part is only visible to the Java Nucleus. A page contains one or more cells where for each cell the contents is preceded by a header pointing to the public information. The private information is obtained by hashing the page frame number to get the per page information which contains the private cell data. The private cell data contains pointers to the public data for all protection domains that share this cell. When a cell is shared between two or more protection domains the pointer in the header of the cell refers to public cell information stored in the private domain specific portion of the virtual address space. The underlying physical pages in this space are different and private for each protection domain.

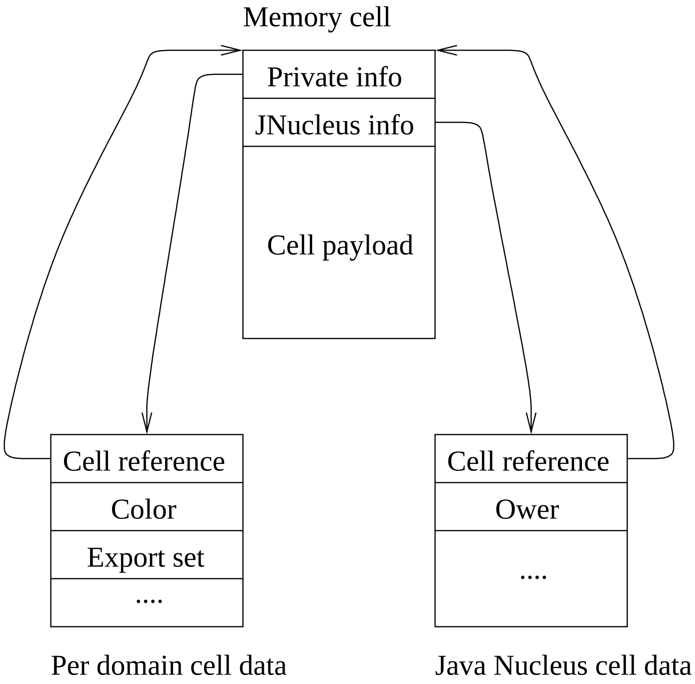


Figure 5.18. Garbage collection cell data structure. A cell consists of the actual, possibly shared, payload and a separate private per domain structure describing the current color and export set. The cell ownership information is kept separately by the Java Nucleus.

To amortize the cost of garbage collection, our implementation stores one or more cells per physical memory page. When all the cells are free, the page is added to the free list. As stated earlier, each protection domain has three memory pools: an execute-only pool, a read-only pool, and a read-write pool. Cells are allocated from these pools depending on whether they contain executable code, constant data, or volatile data. When memory becomes really tight, pages are taken from their free lists, their virtual pages are unmapped, and their physical pages returned to the system physical page pool. This allows them to be re-used for other domains and pools.

Exposing the color and export set fields requires the garbage collector to be very careful in handling these user accessible data items. It does not, however, reduce the security of our system. The user application can, at most, cause the marker phase to loop forever, cause its own cells that are still in use to be deallocated, or hang on to shared pages. These problems can be addressed by bounding the marker loop phase by the number of in-use cells. Deleting cells that are in use will cause the program to fail eventually, and hanging on to shared pages is not different from the program holding on to the reference.

When access to a cell is revoked, for example as a result of an XMI return, its color is marked grey and it is removed from the receiving domain's export set. This will cause the garbage collector to reexamine the cell and unmap it during the sweep phase when there are no references to it from that particular domain.

To relocate a reference, the Java Nucleus forces the garbage collector to start a mark phase and update the appropriate references. Since the garbage collector is exact, it only updates actual object references. An alternative design for relocation is to add an extra indirection for all data accesses. This indirection eliminates the need for explicit pointer updates. Relocating a pointer consists of updating its entry in the table. This design, however, has the disadvantage that it imposes an additional cost on every data access rather than the less frequent pointer assignment operation and prohibits aggressive pointer optimizations by smart compilers.

The amount of memory per protection domain is constrained. When the amount of assigned memory is exhausted an appropriate exception is generated. This prevents protection domains from starving other domains of memory.

5.2.5. Prototype Implementation

Our prototype implementation is based on Kaffe, a freely available JVM implementation [Transvirtual Technologies Inc., 1998]. We used its class library implementation and JIT compiler and we reimplemented the IPC, garbage collector and thread subsystems. Our prototype implements multiple protection domains and data sharing. For convenience, the Java Nucleus contains the JIT compiler and all the native class implementations. It does not yet provide support for text sharing of class implementations and has a simplified security policy description language. Currently, the security policy defines protection domains by explicitly enumerating the classes that comprise it and the access permissions for each individual method. The current garbage collector is not exact for the evaluation stack and uses a weaker form to propagate export set information.

The trusted computing base (TCB) of our system is formed by the Paramecium kernel, the Java Nucleus, and the hardware the system is running on. The size of our Paramecium kernel is about 10,000 lines of commented header files, and C++/assembler code. The current Java Nucleus is about 27,200 lines of commented header files and C++ code. This includes the JIT component, threads, and much of the Java run-time support. The motivation for placing the JIT in the TCB is that it enables certain performance optimizations which we described in Section 5.2.4. In a system that supports text sharing the Java Nucleus can be reduced considerably.

A typical application of our JVM is that of a web server written in Java that supports servlets, like W3C's JigSaw. Servlets are Java applets that run on the web server and extend the functionality of the server. They are activated in response to requests from a web browser and act mainly as a replacement for CGI scripts. Servlets run on behalf of a remote client and can be loaded from a remote location. They should therefore be kept isolated from the rest of the web server.

Our test servlet is the SnoopServlet that is part of the Sun's Java servlet development kit [SunSoft, 1999]. This servlet inherits from a superclass HttpServlet which provides a framework for handling HTTP requests and turning them into servlet method calls. The SnoopServlet implements the GET method by returning a web page containing a description of the browser capabilities. A simple web server implements the HttpServlet superclass. For our test the web server and all class libraries are loaded in protection domain WS, the servlet implementation is confined to Servlet.

The WS domain makes 2 calls into the Servlet domain, one to the constructor for SnoopServlet object and one to the doGet method implementing HTTP GET. This method has two arguments, the servlet request and reply objects. Invoking methods on these causes XMIs back into the WS domain. In this test a total of 217 XMIs occurred. Many of these calls are to runtime classes such as java/io/PrintWriter (62) and java.lang.StringBuffer (101). In an implementation that supports text sharing these calls would be local procedure calls and only 33 calls would require an actual XMI to the web server. Many of these XMIs are the result of queries from the servlet to the browser.

The number of objects that are shared and therefore relocated between the WS and Servlet domains are 47. Most of the relocated objects are static strings (45) which are used as arguments to print the browser information. These too can be eliminated by using text sharing since the underlying implementation of print uses a single buffer. In that case only a single buffer needs to be relocated. The remaining relocated objects are the result of the HttpServlet class keeping state information.

5.3. Discussion and Comparison

Because of the distinct nature of the two applications presented in this chapter we discuss each of them in turn. First the extensible run-time system for Orca followed by a discussion on our secure Java virtual machine.

Extensible Run Time System for Orca

Orca is a language based distributed shared memory system (DSM) for parallel and distributed programming [Bal, 1989]. Orca differs from other DSM systems [Ahuja *et al.*, 1986; Bershad *et al.*, 1993; Johnson *et al.*, 1995; Keleher *et al.*, 1994] in that it encapsulates shared memory into shared data objects. The language semantics guarantee *sequential consistency* on shared object updates and it is up to the language run-time system to implement this. The current run-time systems do this by providing an efficient object state replication strategy and totally ordered group communication.

In our Orca run-time system (FlexRTS) we wanted to be as flexible as possible and used the same approach as we did for the kernel. That is, FlexRTS consists of a small run-time layer and all components, such as a thread system, network protocols, device drivers, specific shared object implementations, *etc.*, are loaded dynamically and on-demand using the extension mechanisms described in Chapter 2. This means that these modules are more amenable to change and experimentation. In addition to the standard modules, the run-time system also has the ability to use specific implementations for individual shared-objects. This enables many application specific optimizations. For example, the shared object could be implemented as a kernel extension, or it could be used to provide a shared-object implementation with different ordering and reliability semantics. Especially the latter might improve the performance of TSP, for example, since its bounds updates require no ordering and very little reliability other than fairness.

The main focus of our FlexRTS system was to show how to construct a flexible framework for a run-time system using the same design principles as for the kernel. As an example we used Orca and showed how to integrate the extension mechanisms into a run-time system and we discussed some alternative implementations for some modules. One of these modules is a group communication protocol which provides efficient shared object updates by off-loading the sequencer and using active messages [Von Eicken *et al.*, 1992]. The group communication protocol itself is similar to Kaashoek's PB protocol [Kaashoek, 1992] except that the recovery state is shared among all members instead of the sequencer. This off-loads the sequencer and allows a higher utilization. Experiments showed that this protocol performed quite well in our experiments but it is expected that the performance degrades under a heavy load with many clients. That is, under heavy load there is a higher probability of packets getting lost and each lost packet results in a group multicast interrupting each member.

Secure Java Virtual Machine

Our system is the first to use hardware fault isolation on commodity components to supplement language protection by tightly integrating the operating system and language run-time system. In our design we concentrated on Java, but our techniques are applicable to other languages as well (*e.g.*, SmallTalk [Goldberg and Robson, 1983] and Modula3 [Nelson, 1991]) provided they use garbage collection, have well defined interfaces, and distinguishable units of protection. A number of systems provide hardware fault isolation by dividing the program into multiple processes and use a proxy based system like RMI or CORBA, or a shared memory segment for communication between them. Examples of these systems are the J-Kernel [Hawblitzel *et al.*, 1998] and cJVM [Aridor *et al.*, 1999]. This approach has a number of drawbacks that are not found in our system:

1. Most proxy mechanisms serialize the data in order to copy it between different protection domains. Serialization provides copy semantics which are

incompatible with the shared memory semantics required by the Java language.

2. The overhead involved in marshaling and unmarshaling the data is significant compared to on demand sharing of data.
3. Proxy techniques are based on interfaces and are not suited for other communication mechanisms such as instance state sharing. The latter is important for object oriented languages.
4. Proxy mechanisms usually require stub generators to generate proxy stubs and marshaling code. These stub generators use interface definitions that are defined outside the language or require language modifications to accommodate them.
5. It is harder to enforce centralized resource control within the system because proxy mechanisms encourage many independent instances of the virtual machine.

The work by Back [Back *et al.*, 1998] and Bernadat [Bernadat *et al.*, 1998] focuses on the resource control aspects of competing Java applets on a single virtual machine. Their work is integrated into a JVM implementation while our method of resource control is at an operating system level. For their work they must trust the bytecode verifier.

The security provided by our JVM consists of separate hardware protection domains, controlled access between them, and system resource usage control. An important goal of our work was to maintain transparency with respect to Java programs. Our system does not, however, eliminate covert channels or solve the capability confinement and revocation problem.

The confinement and revocation problem are inherent to the Java language. A reference can be passed from one domain to another and revocation is entirely voluntary. These problems can be solved in a rather straightforward manner, but they do violate the transparency requirement. For example, confinement can be enforced by having the Java Nucleus prohibit the passing of references to cells for which the calling domain is not the owner. This could be further refined by requiring that the cell owner should have permission to call the remote method directly when the cell is passed to another domain. Alternatively, the owner could mark the cells it is willing to share or maintain exception lists for specific domains. Revocation is nothing more than unmapping the cell at hand.

In the design of our JVM we have been very careful to delay expensive operations until they are needed. An example of this is the on-demand remapping of reference values, since most of the time reference variables are never dereferenced. Another goal was to avoid cross-protection domain switches to the Java Nucleus. The most prominent example of this is pointer assignment which is a trade-off between memory space and security. By maintaining extra, per protection domain, garbage collector state we perform pointer assignments within the same context, thereby eliminating a large number of cross domain calls due to common pointer assignment operations. The amount of state required can be reduced by having the compiler produce hints about the potential sharing opportunities of a variable.

In our current JVM design, resources are allocated and controlled on a per protection domain basis, as in an operating system. While we think this is an adequate protection model, it might prove to be too coarse grained for some applications and might require techniques as suggested by Back [Back *et al.*, 1998].

The current prototype implementation shows that it is feasible to build a JVM with hardware separation whose Java XMI overhead is small. Many more optimizations, as described in this paper, are possible but have not been implemented yet. Most notable is the lack of instruction sharing which can improve the performance considerably since it eliminates the need for XMIs. When these additional optimizations are factored in, we believe that a hardware-assisted JVM compares quite well to JVM's using software fault isolation.

The security of our system depends on the correctness of the shared garbage collector. Traditional JVMs rely on the bytecode verifier to ensure heap integrity and a single protection domain garbage collector. Our garbage collector allocates memory over multiple protection domains and cannot depend on the integrity of the heap. This requires a very defensive garbage collector and careful analysis of all the attack scenarios. In our design the garbage collector is very conservative with respect to addresses it is given. Each address is checked against tables kept by the garbage collector itself and the protection domain owning the object to prevent masquerading. The instance state splitting according to the Java visibility rules prevents adversaries from rewriting the contents of a shared object. Security sensitive instance state that is shared, and therefore mutable, is considered a policy error or a programming error.

Separating the security policy from the mechanisms allows the enforcement of many different security policies. Even though we restricted ourself to maintaining transparency with respect to Java programs, stricter policies can be enforced. These will break transparency, but provide higher security. An example of this is opaque object reference sharing. Rather than passing a reference to shared object state, an opaque reference is passed. This opaque reference can only be used to invoke methods; the object state is not shared and can therefore not be inspected.

The garbage collector, and consequently run-time relocation, have a number of interesting research questions associated with them that are not yet explored. For example, the Java Nucleus is in a perfect position to make global cache optimization decisions because it has an overall view of the data being shared and the XMIs passed between domains. Assigning a direction to the data being shared would allow fine grained control of the traversal of data. For example, a client can pass a list pointer to a server applet which the server can dereference and traverse but the server can never insert one of its own data structures into the list. The idea of restricting capabilities is reminiscent of Shapiro’s *diminish-grant* model for which confinement has been proven [Shapiro and Weber, 2000].

The Java Nucleus depends on user accessible low-level operating system functionality that is currently only provided by extensible operating systems (*e.g.*, Paramecium, OSKit [Ford *et al.*, 1997], L4/LavaOS [Liedtke *et al.*, 1997], ExOS [Engler *et al.*, 1995], and SPIN [Bershad *et al.*, 1995b]). Implementing the Java Nucleus on a conventional operating system would be considerably harder since the functionality listed above is intertwined with hard coded abstractions that are not easily adapted.

Notes

Parts of the FlexRTS sections in this chapter were published in the proceedings of the third ASCI Conference in 1997 [Van Doorn and Tanenbaum, 1997], and the group communication ideas were published in the proceedings of the sixth SIGOPS European Workshop [Van Doorn and Tanenbaum, 1994]. Part of the secure Java virtual machine section appeared in the proceedings of the 9th Usenix Security Symposium [Van Doorn, 2000]. Part of this work has been filed as an IBM patent.

6

Experimental Verification

In this chapter we study some of the quantative aspects of our extensible operating system to get an insight into the performance of our system. For this we ran experiments on the real hardware and on our SPARC architecture simulator (see Section 1.6). In this chapter we discuss the results. We are mostly interested in determining the performance aspects of our extensible system, and how well microbenchmarks of primitives predict the performance of programs that are build on them. That is, we want to determine where time is spent in each part of our system and what fraction of it is due to hardware overhead, extensible system overhead, and the overhead of our particular implementation and whether a microbenchmark result from a lower level explains the result from a microbenchmarks at a higher level.

For our experiments we picked one example from each of the three major subsystems described in this thesis: The extensible kernel, the thread package from the system extensions chapter, and the secure Java virtual machine from the run-time systems chapter. These three examples form a complete application which runs on both the hardware and the simulator and is, therefore, completely under our control. The other applications described in this thesis depend on the network and are therefore less deterministic.

In our opinion, in order to evaluate a system it is absolutely necessary to run complete applications rather than microbenchmarks alone. Microbenchmarks tend to give a distorted view of system because they measure a single operation without taking into effect other operations. For example, the results of a system call microbenchmark might give very good results because the system call code is optimized to prevent cache conflicts. However, when running an application, the performance might not be as advertised because of cache and translation lookaside buffer (TLB) conflicts. Of course, microbenchmarks do give insights into the performance aspects of particular system operations, which is why we discuss our microbenchmarks in Section 6.1. To get a realistic view of a system, complete applications are required, such as our thread system, whose performance aspects are analyzed in Section 6.2, and our secure Java virtual machine, which is analyzed in Section 6.3. The main thread in all these sections is that we try to determine how accurately the results from previous sections predict performance.

6.1. Kernel Analysis

In this section we take a closer look at two different performance aspects of the Paramecium kernel. These are the cost of interprocess communication (IPC) and the cost of invocation chains. These were picked because IPC represents a traditional measure for kernel performance and the invocation chain mechanism forms the basis of the thread system that is discussed in the next section. To provide a sense of the complexity of the kernel, we discuss the number of source code lines used to implement it on a our experimentation platform, a Sun (SPARCClassic) workstation with a 50 MHz MicroSPARC processor.

Interprocess Communication

To determine the cost of an IPC we constructed a microbenchmark that raises an event and causes control to be transfered to the associated event handler. This handler is a dummy routine that returns immediately to the caller. We measured the time it took to raise the event, invoke the handler, and return to the caller. In Paramecium an event handler may be located in any protection domain and the kernel. Given this flexibility, it was useful to measure the time it took to raise an event from the kernel to a user protection domain, from a user protection domain to the kernel, and from a user domain to a different user domain. To establish a base line we also measured the time it took to raise an event from the kernel to a handler also located in the kernel. In order to raise an event, we generated a trap; this is fastest way to raise an event and it is a commonly used mechanism to implement system calls. The benchmark program itself consisted of two parts: a kernel extension and a user application. The application uses the extension to benchmark user-to-kernel IPC. The results for all these benchmarks are shown in Figure 6.1.

		Destination	
		Kernel	**User**
Source	**Kernel**	12.3	10.3
	User	9.5	9.9

Figure 6.1. Measured control transfer latency (in µsec) for null IPCs (from kernel-to-kernel, kernel-to-user, user-to-kernel, and user-to-user contexts).

Our experimentation methodology consisted of measuring the time it took to execute 1,000,000 operations and repeat each of these runs 10 times. The result of these runs were used to determine the average running time per operation. At each run we made sure that the cache and translation lookaside buffer (TLB) were *hot*; this means that the cache and TLB were preloaded with the virtual and physical address and the instruction and data lines that are part of the experiment. The added benefit of a hot cache and TLB is that they provide a well-defined initial state for the simulator.

As is clear from Figure 6.1, when raising an event from a user context it makes little difference whether the handler for an event is located in the kernel or another user context. This result is not too surprising, since the MicroSPARC has a tagged TLB and a physical instruction and data cache. A tagged TLB acts as a cache for hardware MMU context, virtual address and physical address associations. Whenever a virtual address is resolved to a physical address, the TLB is consulted before traversing the MMU data structures. Since the hardware context is part of the TLB associations, there is no need to flush the TLB on a context switch. Likewise, the caches are indexed on physical addresses rather than virtual addresses, so they also do not require flushing on context switches.

The code sequence to transfer control from one context to another is identical in all four cases (kernel-to-kernel, kernel-to-user, user-to-kernel, user-to-user), with the exception that transfers to user space will disable the supervisor flag in the program status register and transfers to kernel space will enable it. Despite the fact that the transfer of control code path is exactly the same, a kernel-to-kernel transfer costs more than a user-to-user transfer. This behavior is due to cache conflicts and will become clear when we discuss sources for performance indeterminacy below.

Even though the performance of our basic system call mechanism is only 25% of a Solaris 2.6 getpid call (a Paramecium null IPC is 9.5 µsec and a Solaris getpid is 37 µsec) running on the same hardware, the absolute numbers are not in the range of other extensible systems and microkernels For example, Exokernel has a 1.4 µsecs null IPC on a 50 MHz MIPS [Engler *et al.*, 1995], and L4 achieves a 0.6 µsecs null IPC on a 200 MHz Pentium Pro [Liedtke *et al.*, 1997]. To determine where the time is being spent, we ran the kernel and microbenchmark on our simulator to obtain instruction traces. For brevity we will only look at the instruction trace for the user-to-kernel IPC since it most closely resembles a traditional system call and therefore compares well with other operating systems.

In Figure 6.2 we have depicted the time line for a single null user-to-kernel IPC with the number of instructions executed and their estimated cycle times. The number of instruction execution cycles used by a RISC instruction depends on the type of operation. Hence, we list the instruction count and instruction cycle count for each simulation run. Typically, a memory operation takes more cycles than a simple register move instruction. The cycle times in Figure 6.2 are an estimate of the real cycle times based on the instruction timings in the architecture manual. They do not take into account TLB and cache misses which can seriously impact the performance of memory operations. These times are not specified in the architecture manual and are in fact quite difficult to emulate because the depend on widely varying memory chip timings. The impact of cache misses becomes clear when we add up the estimated cycle counts: 270 cycles at 50 MHz takes 5.4 μsec as compared to the real measurement of 9.5 μsec, which is a 43% difference. Still, since memory operations have higher cycle time than other operations, the estimated cycle times give a good impression of where the time is being spent. The only way to obtain more accurate instruction cycle counts at this level would be to use an in-circuit emulator (ICE).

The event invocation path, which is the code that handles the trap and transfers control to the event handler, takes up 58% of the instructions and 59% of the cycles in the entire null IPC path. Half of these cycles are spent in saving the contents of the global registers to memory and in register window management. The former consist purely of memory operations, the latter consist of only three memory operations. Most of the register window code is used to determine whether exceptional cases exist that require special processing. In our null IPC path none of these cases occur, but we still have to check for them. The other parts of the event invocation code consist of finding the event handler, maintaining the invocation chain, marking the handler as active, creating a new call frame, setting the supervisor bit in the program status register, switching MMU contexts, and returning from the trap which transfers control back to the microbenchmark function. Setting the current MMU context takes little time, only 3 cycles, because the code is aware that the kernel is mapped into every protection domain. Switching to a user context does require setting the proper MMU context which adds a small number of cycles.

When control is transfered to the microbenchmark function, it immediately executes a return instruction which returns to the event return trap. This is all part of the call frame which was constructed as part of the event invocation.

The event return path, the code that handles the event return trap and transfers control back to the event invoker, accounted for about 40% of the instructions and 38% of the cycles of the entire null IPC path. Again, most of the time is spent restoring the global registers from memory and in register window handling. Other operations that are performed during an event return are: maintaining the invocation chain, freeing the event handler, restoring the context, and clearing the supervisor bit. Two other operations which are of interest, are clearing the interrupt status and restoring the condition codes. These are not necessary for the null IPC, but in the current implementation the IPC return path and the interrupt return path share the same code.† Even though we are returning to user space, there is no need to clear the registers since the event return mechanism recognized that the register window used for the return is still owned by the original owner.

†The current event invoke and return code is not optimal. The initial highly optimized code managed to obtain a null IPC timing of 6.8 μsec, but the code became convoluted after adding more functionality. After that no attempt was made to reoptimize.

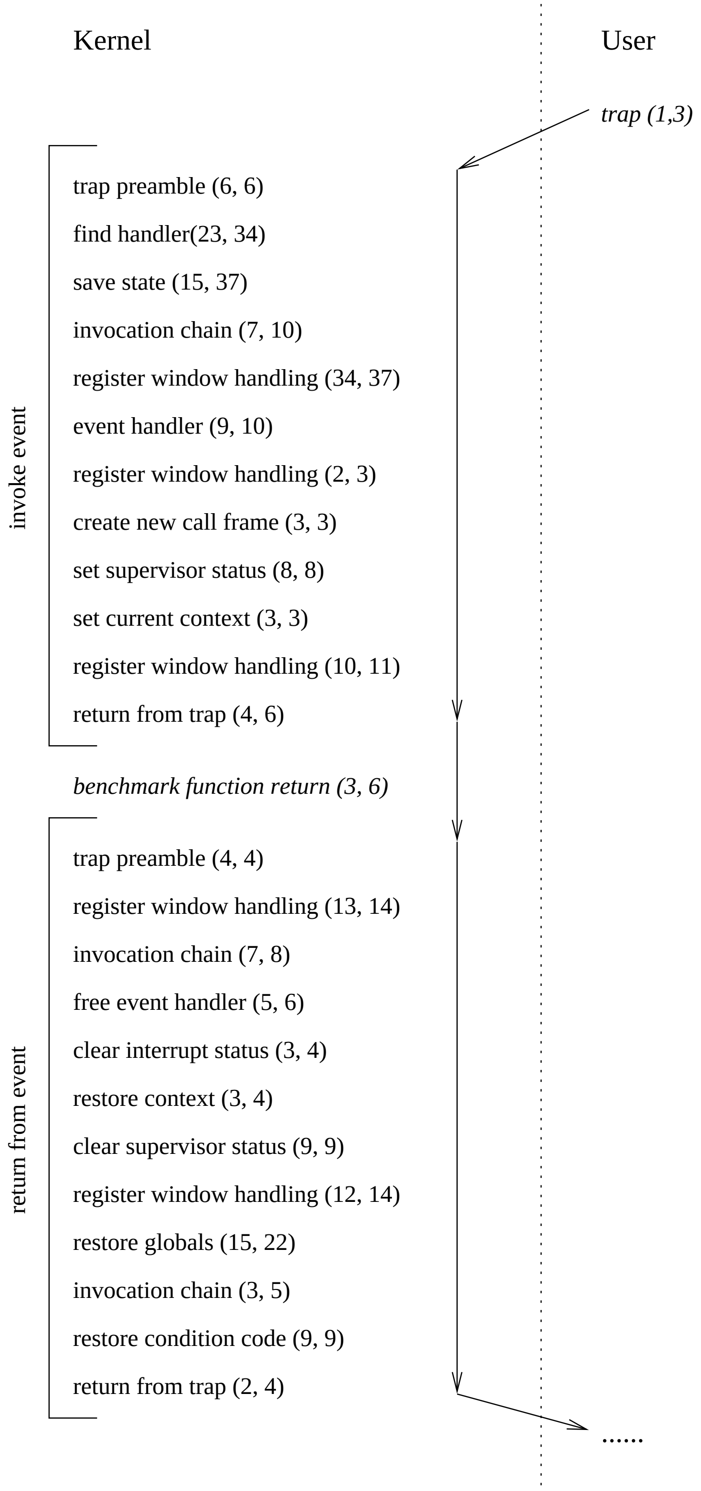

Kernel
User
trap (1,3)
invoke event
trap preamble (6, 6)
find handler(23, 34)
save state (15, 37)
invocation chain (7, 10)
register window handling (34, 37)
event handler (9, 10)
register window handling (2, 3)
create new call frame (3, 3)
set supervisor status (8, 8)
set current context (3, 3)
register window handling (10, 11)
return from trap (4, 6)
benchmark function return (3, 6)
return from event
trap preamble (4, 4)
register window handling (13, 14)
invocation chain (7, 8)
free event handler (5, 6)
clear interrupt status (3, 4)
restore context (3, 4)
clear supervisor status (9, 9)
register window handling (12, 14)
restore globals (15, 22)
invocation chain (3, 5)
restore condition code (9, 9)
return from trap (2, 4)
......

Figure 6.2. Simulated instruction time line for a null user-to-kernel IPC. The values in the parenthesis indicate the number of simulated instructions in an instruction block and the estimated number of execution cycles.

The main reason why our event invocation and return mechanism reduces a system call by 75% when compared to Solaris is that we aggressively reduced the number of register window saves. On Solaris, all register windows are stored to memory at a system call and one window is filled on a return, requiring further register window fill traps by the user applications. In our case, only the global and some auxiliary registers are stored. During an event we mask the register windows belonging to the user as invalid, and these register windows are marked valid again, upon return from the event.

The only problem with the microbenchmarks results discussed above is that they can be misleading. Microbenchmarks typically measure only one specific operation under very controlled and confined circumstances. Typical applications, on the other hand, run in a much more chaotic environment with lots of interactions with other applications. As a result, applications rarely see the performance results obtained by microbenchmarks. The reasons for this performance indeterminacy are:

- *Register window behavior.* Our null IPC microbenchmarks showed very good results but, as became clear from the instruction trace, they did not have to spill or fill any register windows. Most applications do require register window spill and fills which add a considerable performance overhead not indicated by the microbenchmark.

- *Cache behavior.* A microbenchmark typically entirely fits into the instruction and data caches. Applications tend to be much bigger and usually run concurrently with other applications. Therefore the number of cache conflicts is much higher for a normal application than for a microbenchmark. These conflicts result in frequent cache line reloads and add a considerable performance penalty. This behavior is not captured by microbenchmarks.

- *TLB behavior.* Likewise, a TLB has a limited number of entries, 32 in our case, and the TLB working set for a microbenchmark typically fits into these entries. Again, normal applications are bigger and tend to use more kernel services. This results in a bigger memory footprint which in turn requires more TLB entries and therefore has a higher probability to trash the cache. Microbenchmarks typically do not take TLB trashing into account.

The impact of these three performance parameters can be quite dramatic. To investigate this we ran the IPC benchmark, but varied the calling depth by calling a routine recursively before performing the actual IPC. To determine the base line for this benchmark we ran the test without making the actual IPC. These base line results are shown in Figure 6.3 where we measured the time it took to make a number of recursive procedure calls to C++ functions. We ran this benchmark once as an extension in the kernel and again as a user application. As is clear from the figure, the first 5 procedure calls have a small incremental cost which represents the function prologue and epilogue that manages the call frame. However, beyond that point the register window is full and requires overflow processing to free up a register window. The cost of these adds up quickly. There is slight difference between register window overflows in the kernel and user contexts. This is caused by clearing the new register window for user applications to prevent the accidental leaking of sensitive information, such as resource identifiers, from the kernel or between two user protection domains.

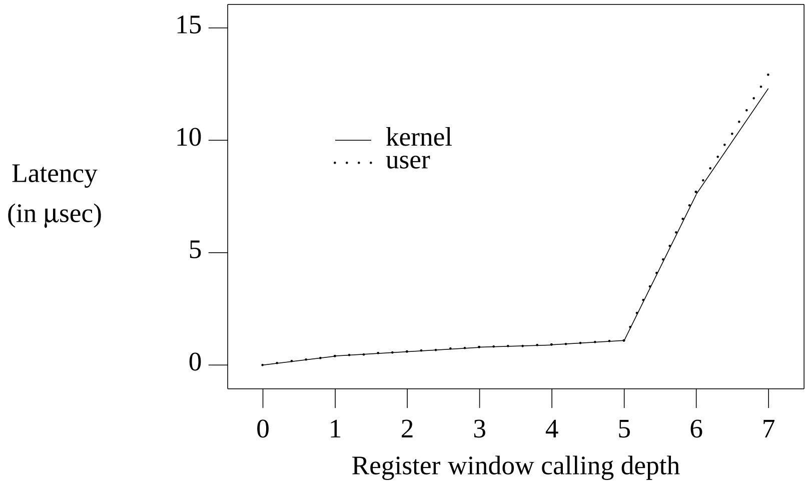


Figure 6.3. Measured register window handling overhead (in µsec) for null procedure calls made in the kernel and from a user application.

Shown in Figure 6.4 are the results for the IPC benchmark while varying the register window calling depth. We used the benchmark to test kernel-to-kernel, kernel-to-user, user-to-kernel, and user-to-user IPCs. The results for all these benchmarks show a similar trend where the register window overhead increases dramatically after four procedure calls. The reason for this is our register window handling code requires an additional window to manage context transitions. The variations between the different benchmarks are puzzling at first since they all execute essentially the same instruction path. The main difference, however, are the locations of the stack and benchmark functions which might cause TLB and cache conflicts. To test this hypothesis we ran the IPC benchmark on our simulator to obtain an estimate for the number of cache and TLB conflicts. We limited our simulator runs to user-to-kernel and kernel-to-kernel IPCs since they represent the best and worst IPC benchmark timings, respectively.

To estimate the number of cache conflicts we modified our simulator to generate instruction and data memory reference and used them to determine the number of cache conflicts. This simulation is quite accurate for two reasons: It uses the same kernel and microbenchmark binaries that are used for obtaining the measurements, and the memory used by them is allocated at exactly the same addresses as on the real hardware. Our experimental platform, a MicroSPARC, has a 2 KB data cache and 4 KB instruction cache. Each cache contains 256 cache lines, where the data cache lines are 8 bytes long and instruction cache lines are 16 bytes long. Both caches are

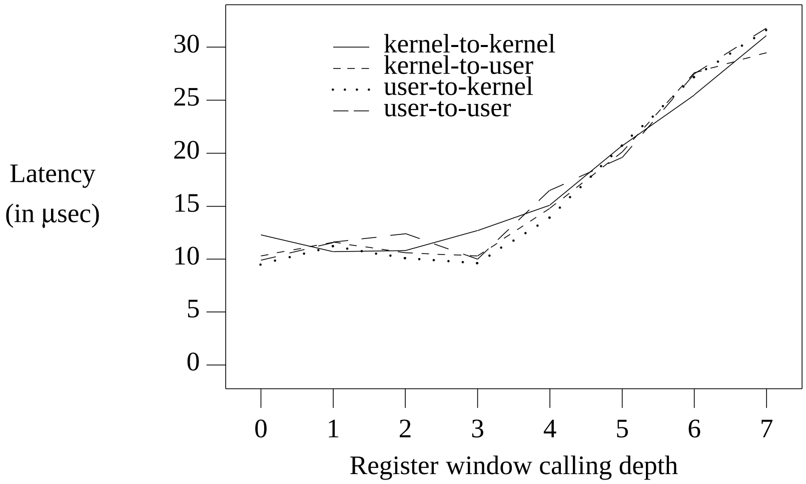


Figure 6.4. Measured control transfer latency (in µsec) for null IPCs depending on register window (from kernel-to-kernel, kernel-to-user, user-to-kernel, and user-to-user contexts).

direct mapped. When discussing cache behavior, we either use the absolute number of cache hits and misses or the hit rate which is traditionally defined as [Hennessy *et al.*, 1996]:

$$\text{hit rate} = \frac{100\ \text{hits}}{(\text{hits}+\text{misses})}\%$$

In Figure 6.5 we show the simulated cache hits and misses for the instruction cache (i-cache) and data cache (d-cache) for user-to-kernel IPCs. We simulated a hot cache, wherein the cache was preloaded with the addresses from a previous run, since that most closely resembles the environment in which our benchmark program was run. To show the effect of register window handling we varied the calling depth. Our simulated cache traces do account for the distortions caused by the clock interrupt, the system watchdog timer, or by the code around the IPC path to keep the statistics. The results can therefore only be used as a performance projection.

As is clear from Figure 6.5, the cache hit rate is quite good for user-to-kernel IPCs. The i-cache hit rate ranges from 99% to 97.7% and the d-cache hit rate ranges from 95.2% to 93.7%. The slight decrease in the hit rate is caused by executing additional instructions to handle register windows and saving the windows to memory.

As became clear from Figure 6.4 there is a slight difference in performance between kernel-to-kernel and user-to-kernel IPCs. When we look at the instruction and data cache misses in Figure 6.6 for kernel-to-kernel and user-to-kernel IPCs it becomes clear that this effect can be partially explained by the higher number of cache misses for a kernel to kernel IPC.

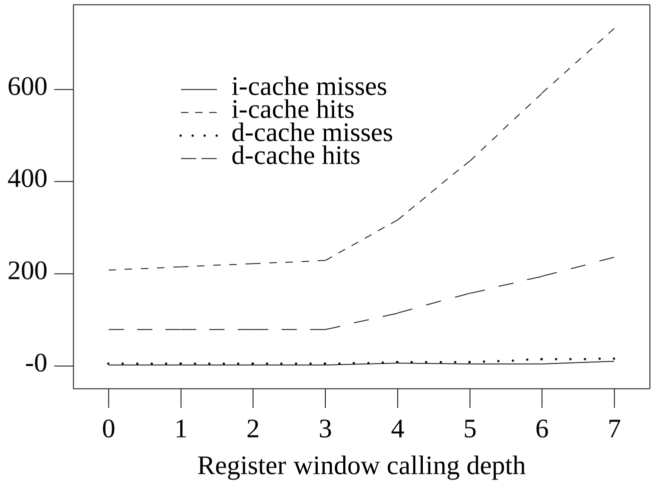


Figure 6.5. Simulated hot instruction and data cache hits and misses for a single null IPC from user-to-kernel while varying the calling depth.

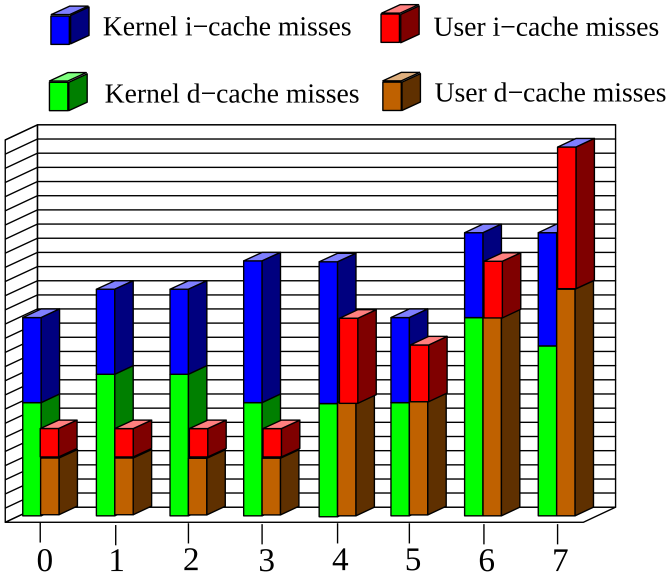


Figure 6.6. Simulated hot instruction and data cache misses for a single null IPC from user-to-kernel and kernel-to-kernel while varying the calling depth.

What is mostly apparent from Figure 6.6 is that the cache behavior is quite erratic even for these small benchmarks running with a hot cache. This is also supported by the fact that the benchmarks gave quite different results when changes were made to the benchmark or the kernel. These could vary as much as 1.4 μsec for the same null IPC benchmark. Because small benchmarks already exhibit such random performance behavior caused by instruction and data cache misses the usefulness of their results is questionable. Granted, the cache in our experimental platform exacerbates the problem of cache conflicts since it is direct mapped and unusually small. Modern caches are typically bigger (32 KB), are set associative with multiple sets, and are typically hierarchical with multiple levels of caches. Hence, the chance for a cache conflict is much smaller than for a one-way set associative cache. Larger caches can therefore handle much more cache conflicts than the small cache on our experimentation platform, but they too are limited, especially when running multiple concurrent applications.

Cache behavior is only one component of performance indeterminacy. Another source is TLB behavior. We used memory traces obtained from the simulator to get an impression of the TLB behavior. Simulating the TLB is quite tricky since a MicroSPARC uses a random replacement algorithm based on the instruction cycle counter. That is, whenever a TLB miss occurs, the TLB entry at the current cycle counter modulo the TLB size (which is 32 for a MicroSPARC) is used to store the new entry. As discussed earlier, the cycle counter values obtained from the simulator are only an approximation. It therefore follows that the TLB simulation results are merely an indication of TLB behavior. The TLB simulations revealed that there were no misses for a hot TLB for a null user-to-kernel an kernel-to-kernel IPC. This is not surprising since the benchmark working set consists of 7 pages and these adequately fit within the TLB of a MicroSPARC.

For the remainder of this chapter we will further examine the IPC performance and determine how well the microbenchmark results from this section compare to real application data.

Invocation Chains

Invocation chains are a coroutine-like abstraction implemented by the kernel and form the basis for our thread system, which is discussed below. The chains mechanism is implemented in the kernel because flushing the register windows, a necessary operation to implement chain swapping, requires supervisor privileges on a SPARC processor. In addition, by implementing chains inside the kernel they can be easily integrated with event management.

The main performance aspect of the invocation chain mechanism is that of chain swapping. That is, how much time does it take to save the context of one chain and resume with another. For this we devised a microbenchmark in which we measured the time it took to switch between two different chains within the same context. Our microbenchmark consists of two chains, the main chain and the auxiliary chain, and at each pass the main chain switches to the auxiliary chain which immediately switches back to the main chains. We used the same experimentation methodology as in the previous subsection and measured how long it took to execute one pass based on 10 runs of 1,000,000 passes each. Since each pass contains two swaps with comparable cost, we divided the result by two to obtain the cost for an individual chain swap. The results are summarized in Figure 6.7.

		Auxilary Chain	
		Kernel	User
Main Chain	Kernel	36.3	67.8
	User	67.8	89.4

Figure 6.7. Measured chain swap latency (in μsec) from the main chain to the auxiliary chain, where the main and auxiliary chain are executing both in the kernel, or both in the same user application, or one chain is executing in the kernel and another in a user application.

Swapping between two kernel chains is the minimum sequence of instructions required to change control from one to another chain. It consists of an interface method invocation to a stub which calls the function that implements the actual chain swap. To swap the chain the current context is saved, the chain pointers are swapped, and the new context is restored. Restoring the new context consists of flushing the current register window set, which saves the registers, and reloads one register window from the new context which causes control to be transfered to the new invocation chain. Swapping two user chains requires more work so we analyze this case in more detail below.

The reason why swapping two user chains requires more work is that it involves a method invocation to an interface implemented by the kernel. This is done by using the proxy mechanism as described in Section 3.4.5. This mechanism allows a user to invoke a method of an interface after which control is transparently transfered to the kernel where the interface dispatch function calls the actual method. In Figure 6.8 we show the call graph of the functions and traps involved in swapping between two user chains. As in the previous section, we used our simulator to obtain an instruction trace and we counted the number of instructions and simulated cycles during a user-to-user chain swap

A user-to-user chain swap executes 1345 instructions and 1682 simulated cycles which takes a simulated 33.6 μsec. As in the previous section, these simulation results do not take cache or TLB misses into effect. This is clear from the measured performance of 89.4 μsec. This mismatch is largely due to d-cache misses. The d-cache has a hit rate of 51.6% The number of i-cache misses on the other hand are minimal. Its hit rate is 87.3%. The simulation showed no TLB misses during a chain swap, which makes sense since the working set for this benchmark, 8 pages, fits comfortably in the TLB.

If we look at the call graph in Figure 6.8, we see that about 59% of the instructions and 62% of the cycles are due to switching chains (disable interrupts, save state, restore state, and enable interrupts). The interface handling overhead (proxy interface,

```
Call: swap chain method (23, 25)
    Trap: invoke event (147, 178)
    Call: interface dispatcher (78, 82)
        Call: chain swap stub (22, 27)
            Call: disable interrupts (40, 49)
            Call: get chain id (45, 50)
            Call: swap chains (43, 55)
                Call: save state (4, 7)
                Call: restore state (7, 9)
                    Trap: register window overflow (61, 84)
                    Call: flush register windows (38, 39)
                        Trap: register window overflow (61, 84)
                        Trap: register window overflow (61, 84)
                        Trap: register window overflow (61, 84)
                        Trap: register window overflow (61, 84)
                        Trap: register window overflow (61, 84)
                        Trap: register window overflow (61, 84)
                        Trap: register window underflow (57,69)
                    Trap: register window underflow (57, 69)
                Call: enable interrupts (25, 31)
            Trap: register window underflow (57,69)
        Trap: register window underflow (57,69)
    Trap: register window underflow (57,69)
    Trap: event return (102, 126)
Trap: register window underflow (59, 71)
```

Figure 6.8. Call graph of a user to user chain swap indicating traps and procedure calls. The values in the parenthesis indicate the number of simulated instructions in an instruction block and the estimated number of execution cycles.

method dispatch and stub method), which accounts for 22% of the instructions and 20% of the cycles, represents the cost introduced by our extensible system. The user-to-kernel transition accounts for about 19% of the instructions and 18% of the cycles.

The question we set out to answer was whether microbenchmark results were representative for application results, in this case user-to-user chain swap performance results. The difference between user-to-user and kernel-to-kernel chain swaps is 53.1 μsec which comprises 59% of the total performance. This time is spent in the IPC, interface dispatch mechanism and a register window underflow routine to handle a sub-routine return. All the other code, invoking the interface method, calling the swap chain stub, saving and restoring state is similar in both cases. At first glance, one would assume the cost of a user-to-user chain swap to be equal to a kernel-to-kernel chain swap plus the overhead for the IPC and interface dispatching. This would lead to the conclusion that 59% of the time is spent in 29% of the executed instructions of which, according to the IPC benchmark, only 9.5 μsec is spent by the IPC and thus the remaining 43.6 μsec would be spent in the interface dispatching code. This is obviously not right since the number of instructions and cycles used by the interface dispatch code are dwarfed by those of the IPC path code. Instead, when we look more closely at the simulated instruction and memory traces we notice two additional register window overflows and one register window underflow between kernel-to-kernel and user-to-user chain swaps. Our cache simulations further revealed that a kernel-to-kernel chain swap has a much higher d-cache hit rate of 72.5%, as opposed to a d-cache hit rate of 51.6% for a user-to-user chain swap. I-cache hit rates were high in both cases, and the TLB simulation revealed no TLB misses. This is not too surprising since the small working set of 8 pages fits comfortably in the TLB. Without an in-circuit emulator (ICE) it is difficult to further refine this performance analysis. However, it is clear that the intuitive approach fails and that the IPC microbenchmark is not a good indicator for the expected user-to-user chain swap performance.

Implementation Complexity

The design guideline for the kernel was to remove everything that is not necessary to preserve the integrity of the system. The result was a small base kernel of about 10,000 lines of source code. As shown in Figure 6.9 the kernel consists of about 70% commented C++ source code (headers and program code) and 27% commented SPARC assembly language source code. The high percentage of assembly language source code is due to the register window handling code, which cannot be written in a high-level language. Most the C++ code deals with implementing event management, virtual and physical memory management, name space management, and device allocation management. About 45% of the source code is machine and architecture dependent. The remainder is machine independent and should be portable to different architectures and platforms. A small number of ODL files define the bindings for the interfaces exported by the kernel and are machine independent. The compiled kernel occupies about 100 KB of instruction text and about 40 KB of initialized data.

6.2. Thread System Analysis

In this section we will be looking at the most important performance characteristic of our unified migrating thread package, the thread context switch cost, and relate the results to the microbenchmarks which we discussed in the previous section. Where possible, we compare and explain the results of similar microbenchmarks that we ran on Solaris 2.6 on the same hardware. Finally, we will discuss the implementation com-

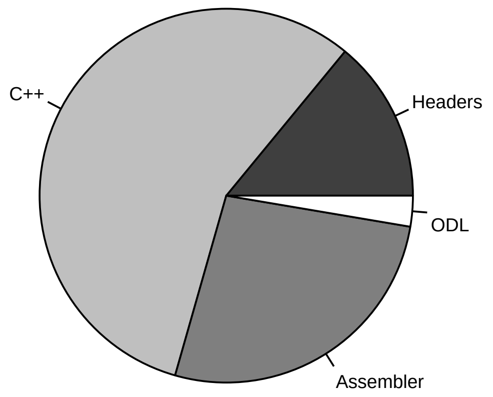


Figure 6.9. Distribution for the lines of source code in the kernel.

plexity of the thread package by looking at the number of source code lines required for its implementation.

Thread Context Switches

One of the most important performance metrics for any thread package, besides the cost of synchronization operations, is the time it takes to switch from one thread context to another. That is, the time necessary to suspend one thread and resume another. Lower thread context switch times are desirable since they enable higher concurrency. To measure the cost of a thread context switch, we devised a microbenchmark that is similar to the microbenchmark for invocation chains which we discussed in the previous section. The microbenchmark runs as a kernel extension or user application and creates an auxiliary thread that repeatedly calls yield to stop its own thread and reschedule another runnable thread. The main thread then measures the performance of itself executing 1,000,000 thread yield operations for 10 runs. When the main thread executes a yield, the auxiliary thread is scheduled which in turn immediately yields, which causes the main thread to be scheduled again. This microbenchmark results in 2,000,000 thread switches per run and the cost per thread switch is shown in Figure 6.10 together with the results of a similar test on Solaris 2.6 using POSIX threads [IEEE, 1996].

We ran the thread context switch microbenchmark in three configurations:

1) As a kernel extension on Paramecium where the two threads were running in the kernel. The thread package and counter device were colocated with the benchmark in the kernel's address space.

Operating System	Thread switch
Paramecium (kernel)	64.2
Paramecium (user)	117.1
Solaris 2.6 (user)	83.5

Figure 6.10. Measured thread context switch time (in μsec) for running the microbenchmark as a Paramecium kernel extension, user application, and as a Solaris 2.6 application.

2) As a user application on Paramecium where the two threads were running in the same user context. Here too the thread package and counter device were colocated with the benchmark in the user's address space.

3) As a user application on Solaris 2.6 where the two threads were running in same user context. The thread package of Solaris is a hybrid thread package in the sense that user-level threads can be mapped onto kernel threads. In our case this mapping did not occur since the two threads did not make any system calls to the kernel.

The low performance numbers for Paramecium in Figure 6.10 for a user-to-user thread context switch are intriguing since a thread context switch is essentially a chain swap. For a chain swap we achieved 36.3 μsec from within a kernel extension and 89.4 μsec from a user application. The latter number is more in line with the measured performance for a Solaris 2.6 thread context switch which essentially consists of a setjmp and longjmp operation (see ANSI C standard [International Standard Organization, 1990]). The Solaris longjmp operations requires a register window flush for which it traps to a fast path into the kernel. Paramecium uses a regular method invocation instead, and performs part of the swapping in the kernel, However, that does not completely explain the 33.6 μsec difference between a Paramecium and Solaris thread switch. Hence, we proceed to determine why a Paramecium user-to-user thread context switch is so expensive as compared to Solaris.

To figure this out, we used the simulator to obtain the call graph shown in Figure 6.11 which shows the calls and traps during one thread switch between two threads executing in user space. Yielding a thread consists of calling the yield method through the thread interface. This will cause control to transfered to the yield stub which sets a global lock, which is used to prevent race conditions, and calls the scheduler. The scheduler determines what to do next, and in this case it consists of resuming the auxiliary thread. To do this, the scheduler leaves the critical region by releasing the global lock and invokes the chain swap method implemented by the kernel. This will cause another thread to be activated, which will then return to the scheduler, which in turn acquires the global lock to enter the critical region. It will update some data structures and return to the yield stub routine, which releases the global lock and returns to the caller.

```
Call: yield method (14,16)
    Call: yield stub (12,14)
        Call: enter critical region (8,11)
        Call: scheduler (52,60)
            Call: leave critical region (8,11)
            Call: swap chain method (1375,1743)
                (see Figure 6.8)
            Call: enter critical region (16,22)
        Trap: window underflow (59,71)
        Call: leave critical region (16,22)
    Trap: window underflow (59,71)
Trap: window underflow (59,71)
```

Figure 6.11. Call graph of a thread switch between two user threads. The values in the parenthesis indicate the number of simulated instructions in an instruction block and the estimated number of execution cycles.

If we look more closely at the number of simulated instructions and cycle counts in Figure 6.11 it becomes that most important cost component is the swapping of two chains, which takes about 82% of the instructions and 83% of the simulated cycles. The code path for kernel-to-kernel thread context switches is exactly the same with the exception that they do not require a user-to-kernel IPC to invoke the chain swap method. If we take the thread context switch costs from Figure 6.10 and subtract the cost for a chain swaps from Figure 6.7 we notice that the additional overhead for kernel-to-kernel thread switches is 27.9 μsec and for user-to-user thread switches it is 27.7 μsec. While deceiving, this is not the overhead caused by our thread package. To determine this, we modified the thread package slightly and commented out the call to the swap chain method in the scheduler and ran the microbenchmark again. This revealed that the additional overhead is merely 7.3 μsec per thread yield call when we ran it as a kernel extension and 9.2 μsec per thread yield call when we ran it as a user application.

Although it is hard to be conclusive about where the time is being spent without an in-circuit emulator, it appears that the thread package incurs a significant overhead for a simple yield operation that cannot be explained by simply looking at the microbenchmark results. For example, if we use the microbenchmark results to estimate a user-to-user thread switch, we would arrive at 89.4 + 9.2 = 98.6 μsec per thread context switch, which does not account for the additional 18.5 μsec we measured. Again we notice that the microbenchmark results are not a good indicator for the actual measured results.

To explain where the additional 18.5 μsec is spent we returned to our simulator for instruction and memory traces. From the instruction traces it became clear that both kernel-to-kernel and user-to-user thread context switches execute two additional window overflows and underflows. This accounts for an approximate additional 9.4 μsec for kernel-to-kernel and 10.4 μsec for user-to-user thread context switches when compared to a pure chain swap. The additional time appears to be spent handling cache misses: A kernel-to-kernel thread context switch results in an i-cache hit rate of 98.3% and a d-cache hit rate of 85%. A user-to-user thread context switch resulted in an i-cache hit rate of 95% and a d-cache hit rate of 73.4%. The TLB simulation showed no TLB misses, which is understandable given that the working set, 11 and 12 page for kernel-to-kernel and user-to-user thread switches, respectively, fits comfortably within the TLB.

Implementation Complexity

The thread package consists of about 3000 lines of source code of which most, 94% (see Figure 6.12), is commented C++ source code (headers and program code). The remaining source code consists of an ODL file, which specifies the bindings for the interfaces exported by the thread package, and a very small assembly language routine that assists in managing call frames. Call frames are the mechanism for linking together procedure calls and are inherently machine dependent. Overall, about 4.5% of the source code is machine dependent, this consists of the assembly language code and a header file containing machine specific in-line functions for implementing an atomic exchange.

6.3. Secure Java Run Time System Analysis

In this section we look at some of the performance aspects of a nontrivial Paramecium application: The secure Java™ virtual machine. Our prototype Java Virtual Machine (JVM) implementation is based on Kaffe, a freely available JVM implementation [Transvirtual Technologies Inc., 1998]. We used its class library implementation and JIT compiler. We reimplemented the IPC, garbage collector, and thread subsystems. Our prototype implements multiple protection domains and data sharing. For convenience, the Java Nucleus, the TCB of the JVM, contains the JIT compiler and all the native class implementations. It does not yet provide support for text sharing of class implementations and has a simplified security policy description language. Currently, the security policy defines protection domains by explicitly enumerating the classes that comprise it and access permissions for each individual method. The current garbage collector is not exact for the evaluation stack and uses a weaker form to propagate export set information.

For this section we used our own IPC microbenchmarks rather than existing ones like Caffeine Marks. The reason for doing so is that many existing benchmarks depend

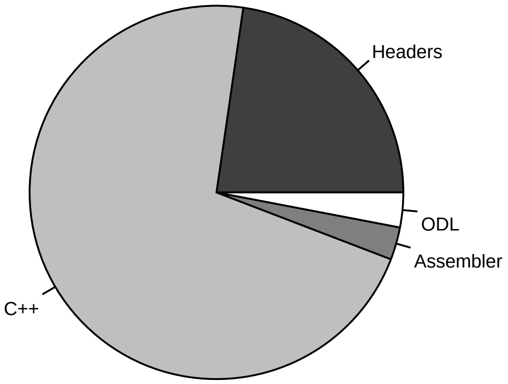


Figure 6.12. Distribution for the lines of source code in the thread system.

on the availability of a windowing system, something Paramecium does not yet support. Existing tests that do not require windowing support are often simple tests that do not exercise the protected sharing mechanisms of the Java Nucleus.

Cross Domain Method Invocations

To determine the cost of a Java cross domain method invocation (XMI) using our Java Nucleus, we constructed two benchmarks that measured the cost of an XMI to a null method and the cost of an XMI to a method that increments a class variable. The increment method shows the additional cost of doing *useful* work over a pure null method call. These two methods both operated on a single object instance whose class definition is shown in Figure 6.13. Like all the other benchmarks, each run consisted of 1,000,000 method invocation and the total test consisted of 10 runs.

The measured performance results for a null XMI and increment XMI, together with their intra-domain method invocations, are summarized in Figure 6.14. We ran the benchmarks in three different configurations:

- One where the classes resided in the same protection domain.
- One where the classes were isolated into different protection domain and where the Java Nucleus was colocated with the kernel.
- One where the classes were isolated into different protection domain and where the Java Nucleus was located a separate user protection domain.

As discussed in Section 5.2.3, colocating the Java Nucleus in the kernel does not reduce the security of the system since both the kernel and the Java Nucleus are considered part of the TCB. However, the configuration where it runs as a separate user

```
class TestObject {
    static int val;

    public void nullmethod()
    {
        return;
    }

    public void increment()
    {
        val++;
    }
}
```

Figure 6.13. Java test object used for measuring the cost of a null method and increment method XMI.

application is more robust since faults caused by the Java Nucleus cannot cause faults in the kernel. On the other hand, colocating the Java Nucleus with the kernel will improve the performance of the system.

Benchmark	**Intra-domain**	**Cross Domain Invocation**	
	Invocation	**Kernel**	**User**
Null XMI	0.7	49.3	101.8
Increment XMI	1.4	53.1	105.1

Figure 6.14. Measured Java method invocation cost (in μsec) for a null method and an increment method within a single protection domain and a cross domain invocation between two Java protection domains where the Java Nucleus is either colocated with the kernel or in a separate protection domain.

Performing an XMI from one object to another, say from the main program BenchMark to the above mentioned TestObject.nullmethod, involves a number of Paramecium IPCs. When the BenchMark program, which resides in its own protection domain, invokes the nullmethod, the method call will cause an exception trap since that object and its class reside in a different protection domain. This will cause control to be transfered to the Java Nucleus since it handles all exceptions thrown by the protection domains it manages. When handling this exception, the Java Nucleus will first determine whether it was caused by a method invocation and, if so, the Java Nucleus will find the corresponding method descriptor. When the exception was not raised by a call or it does not have a valid method descriptor, the Java Nucleus will raise an appropriate Java exception. Using the method descriptor, the Java Nucleus will check the access control list to determine whether the caller may invoke the requested method. If the caller lacks this access, an appropriate Java security exception is raised. If access is permitted, the Java Nucleus will copy the method arguments, if any, from the caller stack onto a new stack that is used to invoke the actual method. The arguments are copied directly, except for pointers which have to be marked exportable as we discussed in 5.2.4. For efficiency, we use a block copy and a bitmask to denote which words in the block constitute pointer values which are handled separately. When all the arguments are processed, an event is invoked to pass control to the actual method which in this case is nullmethod. The latter requires an explicit, and therefore more expensive, event invocation, because it is impossible to associate a specific stack, on which the arguments are copied, with an exception fault.

In the kernel configuration, where the Java Nucleus is colocated with the kernel, only two IPCs are necessary to implement a single XMI: one IPC from a source domain into the Java Nucleus and one IPC from the Java Nucleus to the target domain. The first IPC takes the fast path by raising a memory exception, the second IPC requires an explicit event invocation. Because of this explicit event invocation the user configuration, where the Java Nucleus is isolated in its own separate protection domain, requires three IPCs: one fast event invocation to the Java Nucleus by raising a memory exception, an explicit event invocation to the target domain that requires an additional IPC to the kernel to perform it.

At first glance, we would assume that the XMI performance for the Java Nucleus colocated with the kernel would be the cost of a fast IPC operation from user space, 9.5 μsec, plus an explicit event invocation from the kernel to the target domain, 23.1 μsec†, and thus be something around 32.6 μsec instead of the measured 72.6 μsec. Similarly, for a Java Nucleus running as a user application we would expect the cost of an XMI to be the cost of calling the Java Nucleus from the source domain, 9.9 μsec, plus an IPC to the kernel for the explicit event invocation, 9.5 μsec, plus the actual transfer to the target domain, 23.1 μsec, totaling 42.5 μsec instead of the measured 116.7 μsec. Again we see that the microbenchmarks from Section 6.1 are bad performance predictors and that the actual measured performance is much higher.

This performance discrepancy between the predicted cost and measured cost for a Java XMI between two protection domains is substantial. For example, in the case of the configuration where the Java Nucleus is colocated in the kernel the difference is about 16.7 μsec. To get an insight into where this additional time was spent we used our simulator to obtain instruction and memory traces for single XMI. From this we generated the call graph shown in Figure 6.15 which depicts a single XMI using the configuration where the Java Nucleus is colocated with the kernel.

Figure 6.15 shows that most of the time for an XMI is spent in the code for cross protection domain transitions: that is, invoke event, invoke handler, handler return, and

†In Section 6.1 we focussed event invocations by generating an exception, we did not discuss similar benchmark results using an explicit event invocation. A single kernel to user event invocation takes 23.1 μsec.

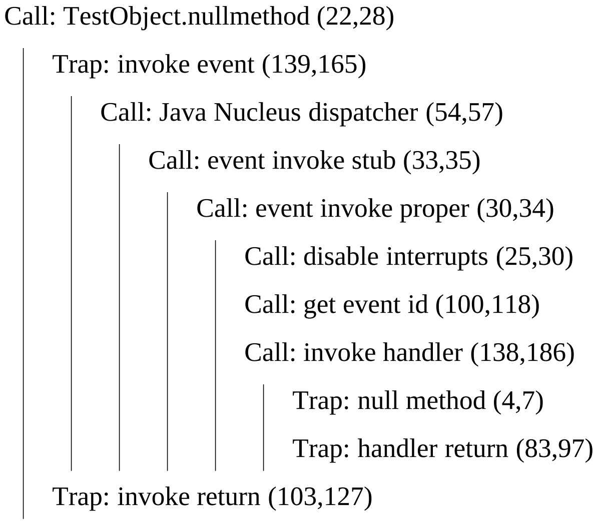

Call: TestObject.nullmethod (22,28)
Trap: invoke event (139,165)
Call: Java Nucleus dispatcher (54,57)
Call: event invoke stub (33,35)
Call: event invoke proper (30,34)
Call: disable interrupts (25,30)
Call: get event id (100,118)
Call: invoke handler (138,186)
Trap: null method (4,7)
Trap: handler return (83,97)
Trap: invoke return (103,127)

Figure 6.15. Call graph of a Java XMI between two Java protection domains where the Java Nucleus is colocated with the kernel. The values in the parenthesis indicate the number of simulated instructions in an instruction block and the estimated number of execution cycles.

event return. These account for about 63% of the instructions and 65% of the simulated cycles. There are no explicit register window overflows and underflows during this XMI, but some windows are implicitly saved and restored during the trap handling which accounts for higher IPC times. The cache simulator revealed a high i-cache hit rate of 95.1%. and a surprisingly high, for the size of the application, d-cache hit rate of 84%. The latter is probably due to the fact that the method did not have any parameters and therefore the Java Nucleus dispatcher did not have to copy arguments or update memory references. The working set was small and revealed no additional TLB misses. Hence, the additional time was probably spent in the implicit register window handling that is part of the IPC code and in handling cache misses.

Implementation Complexity

The current implementation of the Java Nucleus consists of about 27,200 lines of commented header files and C++ code. For convenience this includes the just-in-time compiler and much of the native Java run-time support. Most of the code is machine independent code (57.8%) and it deals with loading class files, code analysis, type checking, *etc.* The just-in-time compiler takes up 20.8% of the source code, and at least 14.2% of the source code consists of implementing the native Java classes such as the thread interface, file I/O, security manager, *etc.* Most of these packages should be implemented outside the Java Nucleus. Finally, only 7.2% of the source code is SPARC dependent, and it includes support code for the just-in-time compiler and stack definitions.

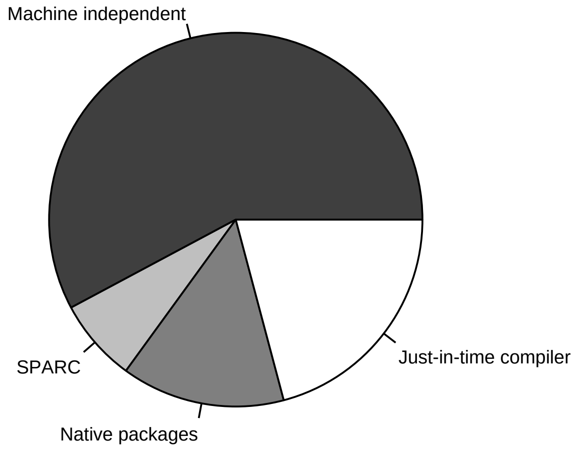


Figure 6.16. Distribution for the lines of source code in the secure Java virtual machine.

6.4. Discussion and Comparison

In this chapter we looked at some of the performance aspects of our extensible operating system kernel, one of its extensions, our thread package, and one of its applications, our secure Java virtual machine. We measured the performance by running a number of (micro) benchmarks on the real hardware and used our SPARC architecture simulator to analyze the results. The main conjecture of this chapter was that microbenchmarks are bad performance predictors because they do not capture the three sources of performance indeterminacy on our target architecture. These are:

- Register window behavior.
- Cache behavior.
- TLB behavior.

These performance indeterminacy sources may have, as we showed in our measurements, serious performance impact for applications and even other microbenchmarks. Granted, our experimentation platform has register windows and an unusually small instruction and data cache which aggravates the problem of cache miss behavior as we showed in our simulations. Although modern machines have bigger and multiway set associative caches, which certainly reduce this problem, these indeterminacy sources still present a source of random performance behavior even on these newer machines.

System level simulators for performance modeling have long been an indispensable tool for CPU and operating system designers [Canon *et al.*, 1979]. Unfortunately, they are not generally available because they contain proprietary information. Academic interest in system level simulators started in 1990 with g88 [Bedichek, 1990] and more recently with SimOS [Rosenblum *et al.*, 1995] and SimICS [Magnusson *et al.*, 1998]. The latter two systems have been used extensively for system level performance modeling. Our simulator started with a more modest goal: The functionally correct emulation of our experimentation platform to aid the debugging of various parts of the operating system. We later extended the simulator to produce memory and instruction traces which were used as input to a cache simulator which emulated the direct mapped 4 KB i-cache and 2 KB d-cache.

In our simulations we tried to be as accurate as possible. We ran the same configuration, the same kernel and benchmark binaries and made sure data was allocated at the same addresses as on the real machine for the actual measurements. Disturbances that were not captured by our simulator included the timer interrupt, the systems watchdog timer, and the exact TLB behavior. All of these require an accurate independent clock, while our simulated clock is driven by the simulated cycle counter which is based on optimistic instruction cost.

7

Conclusions

The central question in this thesis is to determine how useful extensible operating systems are. To determine this, we designed and implemented a new extensible operating system kernel, some commonly used system extensions, and a number of interesting applications in the form of language run-time systems. We showed that we could take advantage of our system in new and interesting ways, but this does not determine how useful extensible systems are in general. It only determines how useful our extensible system is.

In fact, the question is hard to answer. In an ideal extensible system a trained operating system developer should be able to make nontrivial application-specific enhancements that are very hard to do in traditional systems. To test this would require a double experiment in which two groups are given the same set of enhancements where the first group has to implement it on an extensible system and the second on a traditional system. Neither we nor any other extensible systems research group for that matter, has performed this experiment due to the large time and human resource requirements. This would definitely be a worthwhile future experiment.

Even though we cannot answer the question of whether extensible operating systems are useful in general, we can look at our own system and determine which aspects were successful and which not. In the next sections we will analyze, in detail, each individual thesis contribution (summarized in Figure 7.1) and point out its strengths, weaknesses, and possible future work. In turn, we describe the object model, operating system kernel, system extensions, run-time systems, and system performance.

7.1. Object Model

The object model described in Chapter 2 was designed for constructing flexible and configurable systems, and was built around the following key concepts: local objects, interfaces, late binding, classes, object instance naming, and object compositioning.

Topic	Thesis contribution
Object Model	A simple object model for building an extensible systems that consists of interfaces, objects and an external naming scheme.
Kernel	An extensible event-driven operating system. A digital signature scheme for extending the operating system nucleus. A flexible virtual memory interface to support Paramecium's lightweight protection domain model.
System Extensions	A migrating thread package that integrates events and threads and provides efficient cross domain synchronization state sharing. An efficient cross protection domain shared buffer system. An active filter mechanism to support filter based event demultiplexing.
Run-time Systems	An object based group communication mechanism using active messages. An extensible parallel programming system. A new Java virtual machine using hardware fault isolation to separate Java classes transparently and efficiently.
System Performance	A detailed analysis of IPC and context switch paths encompassing the kernel, system extensions, and applications.

Figure 7.1. Main thesis contributions.

Originally the object model was designed to be used in Paramecium and Globe [Van Steen *et al.*, 1999], but the actual implementation and use of the model in these two systems diverged over time. This divergence was caused mainly by the different focus of the two projects. For Paramecium, configurability and size were the most important issues, while for Globe, configurability and reusability were important. In the paragraphs below we discuss the advantages and disadvantages of the object model from a Paramecium perspective and suggest improvements for future work.

Paramecium's flexibility and configurability stem from the use of objects, interfaces, late binding and object instance naming. Separation into objects and interfaces are sound software engineering concepts that help in controlling the complexity of a system. Combining these concepts with late binding and implementing binding to an external object by locating it through the object instance name space turns out to be a very intuitive and convenient approach to build flexible systems. It is also very efficient since the additional overhead for these extensibility features occur mainly at binding time rather than use time. Further enhancements, such as the name space search rules, appear useful in certain situations (see Section 4.1.5) but do require further investigation.

Any object model introduces layering which has an impact on the performance of a system. Traversing a layer, that is, making a procedure call, incurs an overhead that increases with the number of layers introduced. This is not a new problem and has been studied by researchers trying to optimize network protocol stacks. Their automated layer collapsing solutions might be applicable to reduce the layering overhead for our object model.

Paramecium uses classless objects but this is a misnomer: they are actually modules or abstract data types. Paramecium does not use classes because the implementation overhead is considerable, they add complexity rather than help manage complexity and their reusability properties are of limited value. This is not too surprising since, for example, a device driver usually requires only one instance and forcing a class concept onto it does not simplify it. Therefore, a possible future enhancement should be the removal of the class and all associated concepts.

It is also unclear how useful object compositioning is. In the current system composite objects are used more as a conceptual tool than as an actual physical realization. This is no doubt caused by the lack of appropriate tools that assist in the construction of composite objects (tools do exist to build interfaces and local objects which are consequently widely used). The use and construction of composite objects is definitely worth further study.

7.2. Kernel Design for Extensible Systems

In Chapter 3 we described our extensible kernel that forms the basis for the applications and experiments described in this thesis. The kernel forms a small layer on top of the hardware and provides a machine independent interface to it. We attempted to make the kernel as minimal as possible and used the rule that only functions that are crucial for the integrity of system should be included. All others, including the thread system, were implemented as separate components that were loaded into the system on demand. These components can either be loaded into the kernel's address space, provided they have the appropriate digital signature, or in user's address space. To aid experimentation, we took care that almost every component could be loaded either into user of kernel space.

In our experiments we found that colocating modules with the kernel is only necessary in configurations that require sharing. In all other cases, the same functionality could be obtained by colocating the module, such as the thread system or a device driver, with the application in the same user-level's address space. In this thesis we concentrated primarily on application-specific operation systems and as such ran only a single application at a time. In these configurations extending the kernel was hardly ever necessary. Future research should include exploring the usefulness of kernel extensions in an environment with multiple potentially hostile applications.

Our base kernel provides a number of mechanisms that are essential building blocks for extensible applications. Below we will discuss the four main mechanisms to determine how useful they are and what possible future research is required.

Address Space and Memory Management

One of key mechanisms provided by the kernel is address space and memory management. These two mechanisms support Paramecium's lightweight protection domain model in which a single application is divided into multiple protection domains to enforce internal integrity. By decoupling physical and virtual memory management we made it possible for applications to have fine grained control over their virtual memory mappings and shared memory with other protection domains. This enabled applications such as the secure Java virtual machine described in Chapter 5.

Another useful primitive is the ability to give away part of an address space and allow another protection domain to manage it as if it was its own. That is, the ability to create and destroy virtual memory mapping and receive page fault events. This ability forms the basis of our shared buffer mechanism and could be used to implement a memory server. A memory server that manages memory for multiple protection domains is left for future research.

Initially, protection domains are created with an empty name space and it is up to the creator to populate it. That means that a protection domain cannot invoke a method from any interface unless the parent gave it to its child. This is fundamental for building secure systems. A secure operating system based on Paramecium clearly warrants further research. For this it needs to be extended to include mandatory access control to protect resource identifiers, a mechanism to control covert and overt channels between different security levels, and mechanisms to implement strict resource controls.

Event Management

The basic IPC mechanism provided by the kernel is that of preemptive events. Preemptive events are a machine independent abstraction that closely models hardware interrupts, one of the lowest level communication primitives. This particular choice was motivated by the desire to provide fast interrupt dispatching to user-level applications and the desire to experiment with a fully asynchronous system. In addition, from a kernel perspective, preemptive events are simple and efficient to implement. As a general user abstraction they are cumbersome to use, so we created a separate thread package that masked away most of the asynchronous behavior. It still requires applications to lock their data structures conservatively since their thread of control can preempted at any time.

The actual implementation of Paramecium's event handling using register windows was less successful. We spent a lot of time on providing an efficient implementation that reduces the number of register window spills and refills. In retrospect this proved unfruitful for several reasons. First of all, the software overhead for dealing with register windows is highly complex. In fact, the implementation was so complex that we had to developed a full SPARC workstation simulator (described Section 1.6) to debug the register window code. Second, the actual performance gain compared with Probert's [Probert, 1996] register window-less implementation is minimal and his implementation is much less complex. On the other hand, applications do pay a 15% performance penalty [Langendoen, 1997] by not being able to use leaf function optimizations. In addition, when we made the design decisions for our system, compilers that did not use register windows were not readily available and therefore we quickly dismissed a register window-less solution. Finally, implementing an efficient event scheme using register windows is less interesting as they have become an obsolete technology. Except for Sun Microsystems Inc., no one has adopted this technology and even Sun has found other uses for register windows on their SPARC V9 processors [Weaver and Germond, 1994]. These include multithreading support as inspired by MIT's Sparcle processor [Agarwal *et al.*, 1993]. By now, as is underscored by our experience in this thesis, most researchers agree that register windows are a bad idea and that their advantages are mostly overtaken by advances in compile-time and link-time optimizations.

Closely associated with events is the interrupt locking granularity in the kernel. In order to increase the concurrency of the kernel we used fine grained interrupt locking to protect shared data structures in the kernel. In retrospect this was a less fortunate design decision. The reason for this is that the locking overhead is high, especially since each interrupt lock procedure has to generate an interrupt to overcome a potential race condition (see Section 3.4.4) and locks tend to occur in sequences. On the other hand, since the kernel is small and has only short nonblocking operations, there is hardly a need for fine grained interrupt locking. Future work might include exploring the trade-offs between fine grained locking and atomic kernel operations. The later require only one interrupt lock which they hold until the operation is completed.

Name Space Management

The kernel name space manager provides a system-wide location service for interfaces to object instances. A protection domain's interface is populated by its parent who decides what interface to provide to the child. For example, it might choose not to give the child an interface to the console, thereby preventing the child from producing any output. This system-wide name space is a powerful concept that elegantly integrates with kernel extensions. That is, kernel extensions can locate any object instance since the kernel is the parent of all protection domains.

The kernel name space manager automatically instantiates proxy interfaces for objects that are situated in different protection domains. Currently these proxy interfaces are rather rudimentary and limit arguments to the amount that fit into the registers passed from one domain to another. To overcome this limitation and to provide more elaborate and efficient marshaling schemes, a technique such as the one used in Pebble

[Gabber *et al.*, 1999] can be used. How to provide an efficient IPC mechanism with richer semantics is another possible future direction for Paramecium.

Device Management

The Paramecium kernel contains a device manager that is used to allocate devices. Its current policy is simple: the first protection domain to allocate a device obtains an exclusive lock on it that remains active until the device is released. The device manager has some limited knowledge about relationships between devices. For example, when the Ethernet device is allocated on a SPARCClassic it also exclusively locks the SCSI device because of its shared DMA device. Future research directions could include a scheme where multiple protection domains can allocate conflicting devices and use a kernel-level dispatcher, as in the ExoKernel [Engler *et al.*, 1994] or the active filter mechanism, to resolve conflicts.

7.3. Operating System Extensions

The operating system extensions described in Chapter 4 can be divided into two groups. The first group contains the unified migrating thread system and the shared buffer system, both of which support Paramecium's lightweight protection domain model. In this model applications are internally divided into very closely cooperating protection domains to increase their robustness, especially when they are working with untrusted data. The second group consists of the active filter mechanism, which is a way of dispatching events to multiple recipients based on filter predicates. We discuss each of these systems in turn.

Unified Migrating Threads

The unified migrating thread system combines events and threads into a single thread abstraction by promoting events to pop-up threads if they block or take too long. In addition to pop-up threads, the system also supports the notion of thread migration which simplifies the management of threads within a cluster of closely cooperating protection domains. The performance of the thread system, as shown in Chapter 6, is disappointing and is primarily due to the machine's architectural constraints. That is, the thread system has to call the kernel for each thread switch. On different architectures threads can be switched in user space and this would clearly improve the performance of the thread system. However, the thread system still has to call the kernel to switch the event chains since these are the underlying mechanism for thread migration. This kernel call can probably be eliminated by storing the migration state on the user's stack and carefully validating them when returning from an event invocation. However, this is not a perfect solution because it complicates destroying an event chain. An efficient implementation of our thread mechanisms on different architectures certainly merits further study.

Synchronization state sharing is a novel aspect of our thread system and has important applications in closely cooperating protection domains that require frequent synchronization. Our synchronization state sharing technique could be used to optimize Unix's inter-process locking and shared memory mechanisms [IEEE, 1996], but these applications have not been explored.

Network Protocols

The shared buffer system shows the versatility of relinquishing control over part of your virtual address space and giving it to the shared buffer system to manage it for you. Because of this primitive, the implementation of the shared buffer system is simple and straightforward. It only has to keep track of existing buffer pools and map them when requested. Our current buffer scheme provides mutable buffers and expects that every cooperating party is reasonably well behaved. Interesting future research might include exploring a shared buffer system for hostile applications while still providing good performance.

Active Filters

Active filters are an efficient event demultiplexing technique that uses user supplied filter expressions to determine the recipient of an event. Unlike traditional filters, such as packet filters, active filters have access to a part of the user's address space and their evaluation may have side effects. In some sense, active filters are a different extension technique to the one used for the kernel in Chapter 3. Active filters were designed with idea of providing a structured and convenient way to migrate computation from the user into a dispatcher in the same or in a different protection domain or device. Especially migrating computation to an intelligent I/O device is a fruitful area of further research.

7.4. Run Time Systems

In Chapter 5 we presented two different run-time systems that take advantage of the Paramecium extensible operating system kernel and its extension features. The first system, a runtime system for Orca (FlexRTS), shows how to apply application specific extensions to a runtime system. In our second system, a secure Java virtual machine, we show how Paramecium's lightweight protection domain model can be used in a runtime system that deals with potentially mutually hostile applications. Since they are two very different applications we will discuss them separately.

Extensible Run Time System for Orca

In the FlexRTS runtime system we were mostly concerned with the mechanisms required to provide application specific shared-object implementations rather than providing a single solution that fits all. The primary motivation was to be able to relax the ordering requirements for individual shared objects and use different implementation mechanisms such as the active filters described in Chapter 4. Our FlexRTS work raises many questions which are left for future research. For example, in our system the shared-object implementations are carefully hand coded in C++ and it is not clear how these could be expressed in the Orca language themselves. More importantly, how do application-specific shared-object implementations interact with the normal shared objects that enforce a total ordering on their operations? Is there an automatic or annotated way to determine when to relax the ordering semantics? An investigation of both of these questions would be quite interesting.

Secure Java Run-time System

The secure Java Virtual Machine (JVM) presented in Chapter 5 provides a Java execution environment based on hardware fault isolation rather than software fault isolation techniques. This is in sharp contrast with current JVM implementations which are all based on software protection techniques.

The key motivation behind our secure JVM work was to reduce the complexity of the trusted computing base (TCB) for a Java system. Current JVM designs do not make an attempt to minimize the TCB in any way. Rather, they implement the JVM as a single application in a single address space. Our secure JVM implements the JVM in multiple address spaces and limits the TCB to a trusted component, called the Java Nucleus, that handles communication and memory management between the address spaces containing the Java classes. It does this by including some additional management code and, as a result a small increase in latency for method invocations across security boundaries (*i.e.*, the cross protection domain method invocations) and by trading off memory usage versus security. In our opinion this is an acceptable trade-off that is under the control of the security policy.

The prototype implementation described in Section 5.2.5 lacks a number of the optimizations that were presented in other sections of Chapter 5. These would improve the performance of the system considerably and deserve further investigation. The garbage collector used in our system is a traditional mark and sweep collector. These are largely *passé* nowadays and further research in more efficient collectors should be pursued. Such a new collector should also try to optimize the memory used for storing the sharing state. The amount of memory used to store sharing state is currently the greatest deficiency of the system. Given the strong interest in the secure JVM work, it would be beneficial to explore a reimplementation on a traditional UNIX operating system rather than our extensible operating system.

Finally, during the many discussions about this work it became clear that reducing complexity is an amazingly misunderstood property of secure systems. Security is not just a matter of functional correctness, it is also a matter of managing software complexity to reduce the number of implementation errors. Given the current state of software engineering practice, minimizing the TCB is the only successful tool at hand. This is not a new observation; Anderson suggested minimizing the TCB nearly thirty years ago [Anderson, 1972] and the Department of Defense Orange Book series [Department of Defense, 1985] requires minimizing the TCB for B3 and higher systems.

7.5. System Performance

In Chapter 6 we took a closer look at some of the performance issues of our system. We constructed a number of (micro) benchmarks, we measured their performance on our experimental hardware, and we used our simulator to explain the results. The main conjecture in that chapter was that microbenchmarks are typically bad indicators for the end-to-end application performance. We showed that this was true for our benchmarks on our experimentation platform because there are too many sources for performance indeterminacy, such as register windows, cache and TLB behavior, to accurately extrapolate the microbenchmarks results. Of course, our platform exacerbates the problem of performance indeterminacy because it has an exceptionally small cache. Modern machine have much bigger and multi-way set associative caches which have much better cache behavior, but even here extrapolating microbenchmark results is typically not a good indication for system performance because multiple cooperating applications have very erratic cache and TLB behavior. Accurate whole system evaluation is a fruitful area of further research.

7.6. Retrospective

In this thesis we have described the design and some applications of a new operating system. In this section we will briefly look back at the development process and discuss some of the lessons learned.

The hardest part of designing Paramecium was to decide what the abstractions provided by the kernel should be. There was a high-level scientific goal we wanted to achieve, exploring extensible systems, and various technical goals such as a minimal kernel whose primary function is to preserve the integrity of the system and an event driven architecture. The latter was motivated by the desire to experiment with a completely asynchronous system after our experiences with Amoeba [Tanenbaum *et al.*, 1991], a completely synchronous system. We briefly experimented with a number of different primitives but very quickly decided on the ones described in Chapter 3. Once the primitives were set, we started building applications on top of them. Changing the primitives became a constant tension of making tradeoffs between the time it took to work around a particular quirk and the time it took to redo the primitive and all the applications that used it. An example of this is the event mechanism. Events do not block and when there is no event handler available the event invocation fails with an error. The appropriate way of handling this would be to generate an upcall when a thread is about to block and another upcall when a handler becomes available (*i.e.*, use scheduler activation [Anderson *et al.*, 1991] techniques). Timewise, however, it was much more convenient to fix the problem by overcommitting resources and allocate more handlers than necessary. The lesson here is that it is important to decide quickly on a set of kernel abstractions and primitives and validate them by building applications on them but be prepared to change them. Hence it is all right to cut corners, but it should be done sparingly. The most important thing, however, is to realize that an operating system is just a means not the end result.

In retrospect, the decision to use the SPARC platform was unfortunate for multiple reasons. The platform is expensive and therefore not readily available, and the vendor was initially not forthcoming with the necessary information to build a kernel. This resulted in a fair amount of reverse engineering, time that would have been better spent on building the actual system itself. The SPARC platform was only truly understood once the author had written his SPARC architecture simulator. Closely associated with the simulator is the issue of a fast IPC implementation using register windows. The problem here was that a lot of time was spent on optimizing the IPC path too early in the process. That is, the event primitive was still in a state of flux and, more important, we had no application measurements to warrant these optimizations. The lesson here is that it is better to use a popular platform for which the vendor is willing to supply the necessary information, and to wait with optimizing operations until applications indicate that there is a problem that needs to be fixed. It is more important to determine the right primitives first whose design, of course, should not have any inherent performance problems.

Reusing existing implementations turned out to be a mixed blessing. We did not adopt a lot of external code, most of it was written ourself. Some of the code we did adopt, such as the Xinu TCP/IP implement, required a major overhaul to adapt it to our thread and preemptive event model. In the end it was unclear whether the time spent debugging was less than writing a new stack from scratch. Another reason to be wary of using external code is that it is much less well understood and therefore makes debugging harder. The lesson here is that you should only adopt code that closely follows the abstractions provided by the kernel or requires minimal adaptation. Hence, be prepared to write your own code if the external code does not match your abstraction. Do not change your abstractions or primitives to accommodate the external code unless you are convinced the changes have much wider applicability.

For the development of our system extensions and applications we found the absence of high-level abstractions very refreshing. Our low-level abstractions allowed us to rethink basic systems issues and enabled applications and extensions that would be very hard to express on other systems. Of course, our applications were low-level language run-time systems and therefore probably benefited most from our Paramecium abstractions. Perhaps this is the most important lesson of them all, extensible kernels allow us to revisit abstractions that were set almost thirty years ago with Multics [Organick, 1972].

7.7. Epilogue

The goal of this thesis was to show the usefulness of an extensible operating system by studying its design, implementation and some key applications. By doing so, we made the following major research contributions:

- A simple object model that combines interfaces, objects, and an object instance naming scheme for building extensible systems.

- An extensible, event-driven operating system that uses digital signatures to extend kernel boundaries while preserving safety guarantees.
- A new Java virtual machine which uses hardware fault isolation to separate Java classes transparently and efficiently.

In addition, we also made the following minor contributions:

- A migrating thread package with efficient cross protection domain synchronization state sharing.
- An efficient cross protection domain shared buffer system.
- An active filter mechanism to support filter based event demultiplexing.
- An extensible parallel programming system.
- An object based group communication mechanism using active messages.
- A detailed analysis of IPC and context switch paths encompassing the kernel, system extensions, and applications.

In this thesis we demonstrated that our extensible operating system enabled a number of system extensions and applications that are hard or impossible to implement efficiently on traditional operating systems, thereby showing the usefulness of an extensible operating system.

Appendix A
Kernel Interface Definitions

This appendix provides the Interface Definition Language (IDL) specifications for Paramecium's base kernel interfaces as described in Chapter 3. Interface definitions are written in a restricted language consisting of type and constant definitions together with attributes to specify semantic behavior. The IDL generator takes an interface definition and generates stubs for a particular target language. The current generator supports two target languages, C and C++.

The Paramecium interface generator combines two different functions. In addition to generating interface stubs, it also generates object stubs. Objects stubs are generated from an Object Definition Language (ODL) which is a super set of the interface definition language. An object definition describes the interfaces that are exported by the object and their actual method implementations. The object generator aids the programmer in defining new objects.

A.1 Interface Definitions

The interface definition language uses the same declaration definition syntax as defined in the C programming language [International Standard Organization, 1990]. This appendix only describes the IDL specific enhancements and refers to either the C standard or [Kernighan and Ritchie, 1988] for C specific definitions. The syntax definitions are specified in an idealized form of an extended LL(1) grammar [Grune and Jacobs, 1988].

Identifiers are defined similarly to those in the ANSI C standard. In addition to the ANSI C keywords, the interface and nil keywords are also reserved. Comments are introduced either by a slash-slash (‘‘//’’) token and extend up to a new-line character or are introduced by a slash-star (‘‘/*’’) token and extend up to a closing star-slash (‘‘*/’’) token as in ANSI C.

The IDL grammar extends the ANSI C grammar with a new aggregate type representing interfaces. An interface declaration consists of an enumeration of abstract function declarators, describing the type of each method.

interface_type:

interface *identifier interface_def?*

interface_def:

{ [*base_type declarator* **;**]+ **}** [**=** *number*]?

Interfaces are uniquely numbered. This number is chosen by the interface designer and should capture its syntax and semantics since it is used as a primitive form of type checking at interface binding time.

For the programmer's convenience, arguments to method declarations may have auto initializers. These either consist of a scalar value or the keyword nil. The latter represents an untyped null reference.

argument:

base_type abstract_declarator [= [*number* | **nil**]]?

A.2 Base Kernel Interfaces

The types used in the following interface definitions are listed in the table below. Most kernel resources are identified by a 64-bit resource identifier, and are represented by the type resid_t. A naming context is a resource identifier also but for clarity its type is nsctx_t. Physical and virtual addresses are represented by the types paddr_t and vaddr_t, respectively. The ANSI C type size_t denotes the size of a generic object.

Type	Size (in bits)	Description
resid_t	64	Generic resource identifier
nsctx_t	64	Naming context
paddr_t	32	Physical memory address
vaddr_t	32	Virtual memory address
size_t	32	Size (in bytes)

A.2.1 Protection Domains

```
interface context {
    resid_t  create(resid_t name, nsctx_t nsc); // create new context
    void     destroy(resid_t ctxid);         // destroy context
    int      setfault(resid_t ctxid,         // set fault event handler
                 int fault, resid_t evid);
    void     clrfault(resid_t ctxid,         // clear fault event handler
                 int fault);
} = 7;
```

A.2.2 Virtual and Physical Memory

```
interface physmem {
    resid_t  alloc(void);                    // allocate one physical page
    paddr_t  addr(resid_t pp);               // physical address
    void     free(resid_t pp);               // free page
} = 5;

enum accmode { R, RW, RX, RWX, X };
enum attribute { ACCESS, CACHE };

interface virtmem {
    vaddr_t  alloc(resid_t ctxid,            // allocate virtual space
                 vaddr_t vhint,
                 size_t vsize, accmode acc,
                 resid_t *ppids, int npps, resid_t evid);
    void     free(resid_t ctxid,             // free virtual space
                 vaddr_t start, size_t size);
    uint32_t attr(resid_t ctxid,             // set page attributes
                 vaddr_t start, size_t size,
                 attribute index, uint32_t attr);
    resid_t  phys(resid_t ctxid, vaddr_t);   // get physical page
    resid_t  range(resid_t ctxid,            // get range identifier
                 vaddr_t va, size_t size);
} = 6;
```

A.2.3 Thread of Control

```
interface event {
    resid_t  create(resid_t evname);         // create new event
    void     enable(resid_t id);             // enable events
    void     disable(resid_t id);            // disable events
    void     destroy(resid_t evid);          // destroy event
    resid_t  reg(resid_t evname,             // register a new handler
                 resid_t ctxid, void (*method)(...),
                 vaddr_t stk, size_t stksize);
    void     unreg(resid_t evhid);           // unregister a handler
    int      invoke(resid_t evid,            // invoke event
                 void *ap = 0, size_t argsiz = 0);
    void     branch(resid_t evid,            // branch to event
                 void *ap = 0, size_t argsiz = 0);
    vaddr_t  detach(vaddr_t newstk,          // detach current stack
                 size_t newstksize);
} = 4;
```

```
interface chain {
    resid_t  create(resid_t ctxid, vaddr_t pc, // create a new chain
                 vaddr_t stk, size_t stksiz,
                 void *ap = 0, size_t argsiz = 0, vaddr_t sp = 0);
    resid_t  self(void);                       // obtain current chain id
    void     swap(resid_t cid);                // swap to another chain
    void     destroy(resid_t cid);             // destroy chain
} = 8;
```

A.2.4 Naming and Object Invocations

```
interface ns {
    interface soi bind(nsctx_t nsctx, char *name);
    interface soi reg(nsctx_t nsctx, char *name, void *ifp);
    interface soi map(nsctx_t nsctx,
                      char *name, char *file, resid_t where = 0);
    void          unbind(nsctx_t nsctx, void *ifp);
    void          del(nsctx_t nsctx, char *name);
    int           override(nsctx_t nsctx, char *to, char *from);
    nsctx_t       context(nsctx_t nsctx, char *name);

    int           status(nsctx_t nsctx, nsstatus_t *nsbuf);
    nsctx_t       walk(nsctx_t nsctx, int options);
} = 3;
```

A.2.5 Device Manager

```
interface device {
    int      nreg(void);           // # of device registers
    vaddr_t  reg(vaddr_t vhint, int index);
    int      nintr(void);          // # of device interrupts
    resid_t  intr(int index);
    vaddr_t  map(vaddr_t va, int size);
    void     unmap(vaddr_t va, int size);
    int      property(char *name, void *buffer, int size);
} = 9;
```

Notes

The IDL generator was designed and implemented by Philip Homburg. It was adapted by the author to include a C++ target, object definitions (ODL), and some minor extensions to take advantage of the C++ programming language.

Bibliography

Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M., Mach: A New Kernel Foundation for UNIX Development, *Proc. of the Summer 1986 USENIX Technical Conf. and Exhibition*, Atlanta, GA, June 1986, 93-112.

Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B., Yeung, D., D'Souza, G., and Parkin, M., Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors, *IEEE Micro 13*, 3 (June 1993), 48-61, IEEE.

Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers, Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.

Ahuja, S., Carriero, N., and Gelernter, D., Linda and Friends, *Computer 19*, 8 (Aug. 1986), 26-34.

Anderson, J. P., Computer Security Technology Planning Study, ESD-Tech. Rep.-73-51, Vols. I and II, HQ Electronic Systems Division, Hanscom Air Force Base, MA, Oct. 1972.

Anderson, T. E., Lazowska, D. D., and Levy, H. M., The Performance Implications of Thread Management Alternatives for Shared-memory Multiprocessors, *Proc. of the ACM SIGMETRICS International Conf. on Measurement and Modeling of Computer Systems*, Oakland, CA, May 1989, 49-60.

Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M., Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *Proc. of the 13th Symp. on Operating System Principles*, Pacific Grove, CA, Oct. 1991, 95-109.

Anderson, T. E., The Case for Application Specific Operating Systems, *Third Workshop on Workstation Operating Systems*, Key Biscayne, FL, 1992, 92-94.

Andrews, G. R. and Olsson, R. A., *The SR Programming Language: Concurrency in Practice*, Benjamin/Cummings, Redwood City, CA, 1993.

Aridor, Y., Factor, M., and Teperman, A., cJVM: A Single System Image of a JVM on a Cluster, *Proc. of the 1999 IEEE International Conf. on Parallel Processing (ICPP'99)*, Aizu-Wakamatsu City, Japan, Sept. 1999, 4-11.

Arnold, J. Q., Shared Libraries on UNIX System V, in *USENIX Conf. Proc.*, USENIX, Atlanta, GA, Summer 1986, 395-404.

Arnold, K. and Gosling, J., *The Java Programming Language*, Addison Wesley, Reading, MA, Second edition, 1997.

Back, G., Tullman, P., Stoller, L., Hsieh, W. C., and Lepreau, J., Java Operating Systems: Design and Implementation, Tech. Rep. UUCS-98-015, School of Computing, University of Utah, Aug. 1998.

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samuelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., and Woodger, M., Revised Report on the Algorithmic Language Algol 60, 1960.

Bal, H. E., *The shared data-object model as a paradigm for programming distributed systems*, PhD Thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, Holland, Oct. 1989.

Bal, H. E., *Programming Distributed Systems*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., Orca: A Language for Parallel Programming on Distributed Systems, *IEEE Transactions on Software Engineering 18,* 3 (Mar. 1992), 190-205.

Bal, H. E., Bhoedjang, R. A. F., Hofman, R., Jacobs, C., Langendoen, K. G., Rühl, T., and Kaashoek, M. F., Orca: A Portable User-Level Shared Object System, IR-408, Department of Mathematics and Computer Science, Vrije Universiteit, July 1996.

Bal, H. E., Bhoedjang, R. A. F., Hofman, R., Jacobs, C., Langendoen, K. G., and Verstoep, K., *Performance of a High-Level Parallel Language on a High-Speed Network*, Journal of Parallel and Distributed Computing, Jan. 1997.

Barnes, J. G. P., *Programming in Ada*, Addison Wesley, Reading, MA, Third edition, 1989.

Barrera III, J. S., A Fast Mach Network IPC Implementation, *Proc. of the Usenix Mach Symp.,* Monterey, CA, Nov. 1991, 1-11.

Bedichek, R. C., Some Efficient Architecture Simulation Techniques, *Proc. of the Usenix Winter ’90 Conf.*, Washington, D.C, Jan. 1990, 53-63.

Ben−Ari, M., *Principles of Concurrent and Distributed Programming*, Prentice Hall, Englewood Cliffs, NJ, 1990.

Bernadat, P., Lambright, D., and Travostino, F., Towards a Resource-safe Java for Service Guarantees in Uncooperative Environments, *Proc. of the 19th IEEE Real-time Systems Symp. (RTSS’98)*, Madrid, Spain, Dec. 1998.

Berners-Lee, T., Fielding, R., and Frystyk, H., Hypertext Transfer Protocol – HTTP/1.0, RFC-1945, May 1996.

Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M., Lightweight Remote Procedure Call, *Proc. of the 12th Symp. on Operating System Principles*, Litchfield Park, AZ, Dec. 1989, 102-113.

Bershad, B. N., Redell, D. D., and Ellis, J. R., Fast Mutual Exclusion for

Uniprocessors, *Proc. of the Symp. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Sept. 1992, 223-233.

Bershad, B. N., Zekauskas, M. J., and Sawdon, W. A., The Midway Distributed Shared Memory System, *Proc. of Compcon 1993*, San Francisco, CA, Feb. 1993, 528-537.

Bershad, B. N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., and Sirer, E. G., SPIN – An Extensible Microkernel for Application-specific Operating System Services, *Proc. of the Sixth SIGOPS European Workshop*, Wadern, Germany, Sept. 1994, 68-71.

Bershad, B. N., Savage, S., Pardyak, P., Becker, D., Fiuczynski, M., and Sirer, E. G., Protection is a Software Issue, *Proc. of the Fifth Hot Topics in Operating Systems (HotOS) Workshop*, Orcas Island, WA, May 1995, 62-65.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., and Chambers, C., Extensibility, Safety and Performance in the SPIN Operating System, *Proc. of the 15th Symp. on Operating System Principles*, Copper Mountain Resort, CO, Dec. 1995, 267-284.

Bhatti, N. T. and Schlichting, R. D., A System For Constructing Configurable High-Level Protocols, *Proc. of the SIGCOMM Symp. on Communications Architectures and Protocols*, Cambridge, MA, Aug. 1995, 138-150.

Birrell, A., Nelson, G., Owicki, S., and Wobber, E., Network Objects, *Proc. of the 14th Symp. on Operating System Principles*, Ashville, NC, Dec. 1993, 217-230.

Bishop, M., The Transfer of Information and Authority in a Protection System, *Proc. of the Seventh Symp. on Operating System Principles*, Dec. 1979, 45-54.

Black, G., Hsieh, W. C., and Lepreau, J., Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java, *Proc. of the Fourth USENIX Symp. on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000, 333-346.

Boebert, W. E., On the Inability of an Unmodified Capability Machine to Enforce the *-property, *Proc. of the 7th DoD/NBS Computer Security Conference*, Gaithersburg, MD, Sept. 1984, 291-293.

Boehm, B. W., *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ, 1981.

Boehm, H. and Weiser, M., Garbage Collection in an Uncooperative Environment, *Software—Practice & Experience 18*, 9 (1988), 807-820.

Bonneau, C. H., Security Kernel Specification for a Secure Communication Processor, ESD-Tech. Rep.-76-359, Electronic Systems Command, Hanscom Air Force Base, MA, Sept. 1978.

Broadbridge, R. and Mekota, J., Secure Communications Processor Specification, ESD-Tech. Rep.-76-351, Vol. II, Electronic Systems Command, Hanscom Air Force Base, MA, June 1976.

Brooks, F. P., *The Mythical Man−month, Essays on Software Engineering*, Addison Wesley, Reading, MA, 1972.

Burns, A. and Wellings, A., *Real-time Systems and their Programming Languages*, Addison Wesley, Reading, MA, 1990.

Burroughs, *The Descriptor – a Definition of the B5000 Information Processing System*, Burroughs Corporation, Detroit, MI, 1961.

Campbell, R. H., Johnson, G., and Russo, V., Choices (Class Hierarchical Open Interface for Custom Embedded Systems), *ACM Operating Systems Review 21*, 3 (July 1987), 9-17.

Campbell, R. H., Islam, N., Johnson, R., Kougiouris, P., and Madany, P., Choices, Frameworks and Refinement, *Proc. of the International Workshop on Object Orientation in Operating Systems*, Palo Alto, CA, Oct. 1991, 9-15.

Canon, M. D., Fritz, D. H., Howard, J. H., Howell, T. D., Mitoma, M. E., and Rodriquez-Rosell, J., A Virtual Machine Emulator for Performance Evaluation, *Proc. of the Seventh Symp. on Operating System Principles*, Dec. 1979, 71-80.

Cerf, V. G. and Kahn, R. E., A Protocol for Packet Network Interconnection, *TRANS on Communications Technology 22*, 5 (May 1974), 627-641.

Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D., Sharing and Protection in a Single-address-space Operating System, *ACM Transactions on Computer Systems 12*, 4 (Nov. 1994), 271-307.

Cheriton, D. R., The V Distributed System, *Comm. of the ACM 31*, 3 (Mar. 1988), 314-333.

Clark, R. and Koehler, S., *The UCSD Pascal Handbook*, Prentice Hall, Englewood Cliffs, NJ, 1982.

Colwell, R. P., The Performance Effects of Functional Migration and Architectural Complexity in Object-Oriented Systems, CMU, PhD Thesis, Department of Computer Science, CMU, Pittsburgh, PA, Aug. 1985.

Comer, D. E. and Stevens, D. L., *Internetworking with TCP/IP, Volume II: Design, Implementation, and Internals*, Prentice Hall, Englewood Cliffs, NJ, Second edition, 1994.

Common Criteria, Common Criteria Documentation, (available as http://csrc.nist.gov/cc), 2000.

Custer, H., *Inside Windows NT*, Microsoft Press, Redmond, WA, 1993.

Dahl, O. J. and Nygaard, K., SIMULA – An Algol-based simulation language, *Comm. of the ACM 9* (1966), 671-678.

Daley, R. C. and Dennis, J. B., Virtual Memory, Process, and Sharing in Multics, *Comm. of the ACM 11*, 5 (May 1968), 306-312.

Dasgupta, P. and Ananthanarayanan, R., Distributed Programming with Objects and Threads in the Clouds System, *USENIX Computing Systems 4*, 3 (1991), 243-275.

Dean, D., Felten, E. W., and Wallach, D. S., Java Security: From HotJava to Netscape and Beyond, *Proc. of the IEEE Security & Privacy Conf.*, Oakland, CA, May 1996, 190-200.

Dennis, J. B. and Van Horn, E. C., Programming Semantics for Multiprogrammed Computations, *Comm. of the ACM 9,* 3 (Mar. 1966), 143-155.

Department of Defense, Trusted Computer System Evaluation Criteria, DoD 5200.28-STD, National Computer Security Center, Ft. Meade, MD, Dec. 1985.

Des Places, F. B., Stephen, N., and Reynolds, F. D., Linux on the OSF Mach3 microkernel, *Conf. on Freely Distributable Software,* Boston, MA, Feb. 1996.

Deutsch, L. P., Design Reuse and Frameworks in the Smalltalk-80 System, in *Software Reusability, Volume II: Applications and Experience*, Addison Wesley, Reading, MA, 1989, 57-71.

Dijkstra, E. W., *Cooperating Sequential Processes*, Academic Press, New York, 1968.

Dijkstra, E. W., The Structure of the ‘‘THE’’-Multiprogramming System, *Comm. of the ACM 11*, 5 (May 1968), 341-346.

Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F., On-the-fly Garbage Collection: An Exercise in Cooperation, *Comm. of the ACM 21*, 11 (Nov. 1978), 965-975.

Dimitrov, B. and Rego, V., Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms, *IEEE Transactions on Parallel and Distributed Systems 9*, 5 (May 1998), 459-469.

Doligez, D. and Gonthier, G., Portable Unobtrusive Garbage Collection for Multiprocessor Systems, *Proc. of the 21st Annual ACM SIGPLAN Notices Symp. on Principles of Programming Languages*, Jan. 1994, 70-83.

Dorward, S., Presotto, D., Trickey, H., Pike, R., Ritchie, D., and Winterbottom, P., Inferno, *Proc. of Compcon 1997,* Los Alamitos, CA, Feb. 1997, 241-244.

Druschel, P. and Peterson, L., High-Performance Cross-Domain Data Transfer, Tech. Rep. 92-11, Department of Computer Science, University of Arizona, Mar. 30, 1992.

Druschel, P. and Peterson, L., Fbufs: A High-bandwidth Cross Domain Transfer Facility, *Proc. of the 14th Symp. on Operating System Principles*, Ashville, NC, Dec. 1993, 189-202.

Eide, E., Frei, K., Ford, B., Lepreau, J., and Lindstrom, G., Flick: A flexible, optimizing IDL compiler, *Proc. of the ACM SIGPLAN Notices ’97 Conf. on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997, 44-56.

England, D. M., Capability, Concept, Mechanism and Structure in System 250, *RAIRO-Informatique (AFCET) 9* (Sept. 1975), 47-62.

Engler, D., Chelf, B., Chou, A., and Hallem, S., Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions , *Proc. of the Fourth USENIX Symp. on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000, 1-16.

Engler, D. R., Kaashoek, M. F., and O’Toole Jr., J., The Operating Systems Kernel as a Secure Programmable Machine, *Proc. of the Sixth SIGOPS European Workshop,* Wadern, Germany, Sept. 1994, 62-67.

Engler, D. R., Kaashoek, M. F., and O'Toole Jr., J., Exokernel: An Operating System Architecture for Application-Level Resource Management, *Proc. of the 15th Symp. on Operating System Principles*, Copper Mountain Resort, CO, Dec. 1995, 251-266.

Engler, D. R. and Kaashoek, M. F., DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation, *Proc. of the SIGCOMM'96 Conf. on Applications, Technologies, Architectures and Protocols for Computer Communication*, Palo Alto, CA, Aug. 1996, 53-59.

Engler, D. R., VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System, *Proc. of the ACM SIGPLAN Notices '96 Conf. on Programming Language Design and Implementation (PLDI)*, 1996, 160-170.

Esmertec, Jbed Whitepaper: Component Software and Real-Time Computing, White paper, Esmertec, 1998. (available as http://www.jbed.com).

Felten, E., Java's Security History, (available as http://www.cs.princeton.edu/sip/history.html), 1999.

Fitzgerald, R. and Rashid, R. F., The Integration of Virtual Memory Management and Interprocess Communication in Accent, *ACM Transactions on Computer Systems 4*, 2 (May 1986), 147-177.

Ford, B. and Lepreau, J., Evolving Mach 3.0 to a Migrating Thread Model, *Proc. of the Usenix Winter '94 Conf.*, San Francisco, CA, Jan. 1994, 97-114.

Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S., Microkernels Meet Recursive Virtual Machines, *Proc. of the Second USENIX Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996, 137-151.

Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., and Shivers, O., The Flux OSKit: A Substrate for Kernel and Language Research, *Proc. of the 16th Symp. on Operating System Principles*, Saint-Malo, France, Oct. 1997, 38-51.

Fujitsu Microelectronics Inc., *SPARCLite Embedded Processor User's Manual*, Fujitsu Microelectronics Inc., 1993.

Gabber, E., Small, C., Bruno, J., Brustoloni, J., and Silberschatz, A., Building Efficient Operating Systems from User-Level Components in Pebble, *Proc. of the Summer 1999 USENIX Technical Conf.*, 1999, 267-282.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns Elements of Reusable Object-oriented Software*, Addison Wesley, Reading, MA, 1995.

Ghezzi, C. and Jazayeri, M., *Programming Language Concepts*, John Wiley & Sons, New York, NY, Second edition, 1987.

Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, MA, 1983.

Goodheart, B. and Cox, J., *The Magic Garden Explained The Internals of UNIX System V Release 4 and Open System Design*, Prentice Hall, Englewood Cliffs, NJ, 1994.

Gosling, J., Joy, B., and Steele, G., *The Java Language Specification*, Addison Wesley, Reading, MA, 1996.

Graham, I., *Object Oriented Methods*, Addison Wesley, Reading, MA, 1993.

Grune, D. and Jacobs, C. J. H., A Programmer-friendly LL(1) Parser Generator, *Software – Practice and Experience 18*, 1 (Jan. 1988), 29-38.

Guthery, S. B. and Jurgensen, T. M., *Smart Card Developer's Kit*, Macmillian Technical Publishing, Indianapolis, IN, 1998.

Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolter, J., The Performance of μ-Kernel-Based Systems, *Proc. of the 16th Symp. on Operating System Principles*, Saint-Malo, France, Oct. 1997, 66-77.

Habert, S., Mosseri, L., and Abrossimov, V., COOL: Kernel Support for Object-oriented Environments, *Proc. on ECOOP/Object-Oriented Programming Systems, Languages and Applications*, Ottawa, Canada, Oct. 1990, 269-277.

Halbert, D. C. and Kessler, P. B., Windows of Overlapping Register Frames, *CS 292R Final Reports*, June 1980, 82-100.

Handy, J., *The Cache Memory Book*, Academic Press, 1993.

Hardy, N., The KeyKOS Architecture, *ACM Operating Systems Review*, Oct. 1995, 8-25.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D., Protection in Operating Systems, *Comm. of the ACM 19*, 8 (Aug. 1976), 461-471.

Hawblitzel, C., Chang, C., Czajkowski, G., Hu, D., and Von Eicken, T., Implementing Multiple Protection Domains in Java, *Proc. of the 1998 USENIX Annual Technical Conf.*, New Orleans, LA, June 1998, 259-270.

Hennessy, J., Goldberg, D., and Patterson, D. A., *Computer Architecture a Quantitative Approach*, Morgan Kaufmann Publishers Inc., Second edition, 1996.

Hewitt, C., Viewing Control Structures as Patterns of Passing Messages, MIT AI Lab Memo 410, MIT, Dec. 1976.

Hildebrand, D., An Architectural Overview of QNX, *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, Seattle, WA, Apr. 1992, 113-116.

Homburg, P., *The Architecture of a Worldwide Distributed System*, PhD Thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, Holland, Mar. 2001.

Homburg, P., Van Doorn, L., Van Steen, M., and Tanenbaum, A. S., An Object Model for Flexible Distributed Systems, *Proc. of the First ASCI Conf.*, Heijen, The Netherlands, May 1995, 69-78.

Hopcroft, J. E. and Ullman, J., *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, Reading, MA, 1979.

Hsieh, W. C., Kaashoek, M. F., and Weihl, W. E., The Persistent Relevance of IPC Performance: New techniques for Reducing the IPC Penalty, *Proc. Fourth Workshop on Workstation Operating Systems*, Napa, CA, Oct. 1993, 186-190.

Hsieh, W. C., Johnson, K. L., Kaashoek, M. F., Wallach, D. A., and Weihl, W. E., Efficient Implementation of High-Level Languages on User-Level Communication Architectures, MIT/LCS/Tech. Rep.-616, May 1994.

Hutchinson, N. C., Peterson, L. L., Abbott, M. B., and O'Malley, S., RPC in the *x*-kernel: Evaluating New Design Techniques, *Proc. of the 12th Symp. on Operating System Principles*, Litchfield Park, AZ, Dec. 1989, 91-101.

IEEE, Standard for Boot Firmware, 1275, IEEE, Piscataway, NJ, 1994.

IEEE, American National Standards Institute/IEEE Std 1003.1, ISO/IEC 9945-1, IEEE, Piscataway, NJ, 1996.

Ingals, D. H. H., The Smalltalk-76 Programming System: Design and Implementation, *5th ACM Symp. on Principles of Programming Languages,* Tucson, AZ, 1978, 9-15.

International Standard Organization, Programming Language C, 9899, ISO/IEC, 1990.

Jaeger, T., Liedtke, J., Panteleenko, V., Park, Y., and Islam, N., Security Architecture for Component-based Operating Systems, *Proc. of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, 1998, 222-228.

Johnson, K. L., Kaashoek, M. F., and Wallach, D. A., CRL: high-performance all-software distributed shared memory, *Proc. of the 15th Symp. on Operating System Principles*, Copper Mountain Resort, CO, Dec. 1995, 213-226.

Jones, M. B., Interposing Agents: Transparently Interposing User Code at the System Interface, *Proc. of the 14th Symp. on Operating System Principles*, Ashville, NC, Dec. 1993, 80-93.

Jones, R. and Lins, R., *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*, John Wiley & sons, New York, 1996.

Kaashoek, M. F. and Tanenbaum, A. S., Group Communication in the Amoeba Distributed Operating System, *Proc. of the 11th IEEE Symp. on Distributed Computer Systems*, Arlington, TX, May 1991, 222-230.

Kaashoek, M. F., *Group communication in distributed computer systems*, PhD Thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, Holland, 1992.

Kaashoek, M. F., (personal communication), 1997.

Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceño, H., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Janotti, J., and Mackenzie, K., Application Performance and Flexibility on Exokernel Systems, *Proc. of the 16th Symp. on Operating System Principles*, Saint-Malo, France, Oct. 1997, 52-65.

Karger, P. A. and Herbert, A. J., An Augmented Capability Architecture to Support Lattice Security and Traceability of Access, *Proc. of the IEEE Security & Privacy Conf.*, Oakland, CA, Apr. 1984, 2-12.

Karger, P. A., *Improving Security and Performance for Capability Systems*, PhD Thesis, University of Cambridge Computer Laboratory, Cambridge, England, Oct. 1988.

Keleher, P., Cox, A. L., Dwarkadas, S., and Zwaenepoel, W., Tread Marks: Distributed

Shared Memory on Standard Workstations and Operating Systems, *Proc. of the USENIX Winter 1994 Technical Conf.*, Jan. 1994, 115-131.

Keppel, D., Tools and Techniques for Building Fast Portable Threads Packages, Tech. Rep. 93-05-06, University of Washington, CSE, 1993.

Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1988.

Kilburn, T., Edwards, D. B. G., Lanigan, M. J., and Sumner, F. H., One-Level Storage System, *IEEE Transactions on Electronic Computers EC-11*, 2 (Apr. 1962), 223-235.

Korf, R., Depth-First Iterative-Deepening: An Optimal Admissible Tree Search, *Artificial Intelligence*, 1985, 97-109.

Kronenberg, N., Benson, T. R., Cardoza, W. M., Jagannathan, R., and Thomas, B. J., Porting OpenVMS from VAX to Alpha AXP, *Comm. of the ACM 36*, 2 (Feb. 1993), 45-53.

Krueger, K., Loftesness, D., Vahdat, A., and Anderson, T., Tools for the development of application-specific virtual memory management, *Proc. on Object-Oriented Programming Systems, Languages and Applications*, Washington, DC, Oct. 1993, 48-64.

Kung, H. T. and Song, S. W., An efficient parallel garbage collection system and its correctness proof, *IEEE Symp. on Foundations of Computer Science*, 1977, 120-131.

Lamport, L., How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *Transactions on Computers 9*, 28 (Sept. 1979), 690-691, IEEE.

Lampson, B., Pirtle, M., and Lichtenberger, W., A User Machine in a Time-sharing System, *Proc. IEEE 54*, 12 (Dec. 1966), 1766-1774.

Lampson, B., A Note on the Confinement Problem, *Comm. of the ACM 16*, 10 (Oct. 1973), 613-615.

Lampson, B., Protection, *Operating Systems Review 8*, 1 (Jan. 1974), 18-24.

Langendoen, K., (personal communication), 1997.

Lauer, H. C., Observations on the Development of Operating Systems, *Proc. of the Eighth Symp. on Operating System Principles*, Pacific Grove, CA, Dec. 1981, 30-36.

Lea, R., Yokote, Y., and Itho, J., Adaptive Operating System Design Using Reflection, *Proc. of the Fifth Hot Topics in Operating Systems (HotOS) Workshop*, Orcas Island, WA, May 1995, 95-105.

Levy, H. M., *Capability-based Computer Systems*, Digital Press, Bedford, MA, 1984.

Li, K. and Hudak, P., Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems 7*, 4 (Nov. 1989), 321-359.

Lieberman, H., Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Languages, *Proc. on Object-Oriented Programming Systems, Languages and Applications, ACM SIGPLAN Notices 21*, 11 (1986), 214-223.

Liedtke, J., Clans & Chiefs, in *GI/ITG-Fachtagung Architektur von Rechensystemen*, Springer-Verlag, Berlin-Heidelberg-New York, 1992, 294-304.

Liedtke, J., Elphinstone, K., Schönberg, S., Härtig, H., Heiser, G., Islam, N., and Jaeger, T., Achieved IPC Performance (Still The Foundation For Extensibility), *Proc. of the Sixth Workshop on Hot Topics in Operating Systems (HotOS)*, Chatham (Cape Cod), MA, May 1997, 28-31.

Lindholm, T. and Yelin, F., *The Java Virtual Machine Specifications*, Addison Wesley, Reading, MA, 1997.

Lippman, S., *Inside the C++ Object Model*, Addison Wesley, Reading, MA, 1996.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., Abstraction Mechanisms in CLU, *Comm. of the ACM 20*, 8 (1977), 564-575.

Luger, G. F. and Stubblefield, W. A., *Artificial Intelligence and the Design of Expert Systems*, The Benjamin/Cummings Publishing Company, Inc., 1989.

Maeda, C. and Bershad, B. N., Protocol Service Decomposition for High-Performance Networking., *Proc. of the 14th Symp. on Operating System Principles*, Ashville, NC, Dec. 1993, 244-255.

Magnusson, P. S., Larsson, F., Moestedt, A., Werner, B., Dahlgren, F., Karlsson, M., Lundholm, F., Nilsson, J., Stenström, P., and Grahn, H., SimICS/sun4m: A Virtual Workstation, *Proc. of the 1998 USENIX Annual Technical Conf.*, New Orleans, LA, June 1998, 119.

Mascaranhas, E. and Rego, V., Ariadne: Architecture of a Portable Threads System Supporting Thread Migration, *Software Practice and Experience 26*, 3 (Mar. 1996), 327-357.

Massalin, H., *Synthesis: An Efficient Implementation of Fundamental Operating System Services*, PhD Thesis, Department of Computer Science, Columbia University, New York, NY, 1992.

Maxwell, S. E., *Linux Core Kernel Commentary*, The Coriolis Group, 1999.

McKusick, M. K., Bostic, K., and Karels, M. J., *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesley, Reading, MA, 1996.

McVoy, L. and Staelin, C., lmbench: Portable Tools for Performance Analysis, *USENIX 1996 Annual Technical Conf.*, San Diego, CA, Jan. 22-26, 1996, 279-294.

Menezes, A. J., Oorschot, P. C. V., and Vanstone, S. A., *Handbook of Applied Cryptography*, CRC Press, 1997.

Microsoft Corporation and Digital Equipment Corporation, Component Object Model Specification, Oct. 1995.

Milner, R., Tofte, M., and Harper, R., *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1990.

Mitchell, J. G., Gibbons, J. J., Hamilton, G., Kessler, P. B., Khalidi, Y. A., Kougiouris, P., Madany, P. W., Nelson, M. N., Powell, M. L., and Radia, S. R., An Overview of the Spring System, *Proc. of Compcon 1994*, Feb. 1994, 122-131.

Mohr, E., Kranz, D. A., and Jr., R. J. H., Lazy Task Creation: A Technique for

Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, July 1992, 264-280.

Montz, A. B., Mosberger, D., O'Malley, S. W., Peterson, L. L., Proebsing, T. A., and Hartman, J. H., Scout: A Communications-Oriented Operating System, *Proc. of the first USENIX Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994, 200.

Moon, D. A., Genera Retrospective, *Proc. of the International Workshop on Object Orientation in Operating Systems*, Palo Alto, CA, Oct. 1991, 2-8.

Moore, C. H. and Leach, G. C., FORTH - A Language for Interactive Computing, *(internal publication)*, Amsterdam, NY, 1970.

Moore, C. H., FORTH: A New Way to Program a Computer, *Astronomy & Astrophysics Supplement Series 15*, 3 (1974).

Myers, G. J., Can Software for SDI Ever be Error-free?, *IEEE computer 19*, 11 (1986), 61-67.

Necula, G. C. and Lee, P., Safe Kernel Extensions Without Run-Time Checking, *Proc. of the Second USENIX Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996, 229-243.

Nelson, G., *Systems Programming with Modula-3*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, MA, 1972.

Organick, E. I., *A Programmer's View of the Intel 432 System*, McGraw-Hill, New York, 1983.

Otte, R., Patrick, P., and Roy, M., *Understanding CORBA, The Common Object Request Broker Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1996.

Ousterhout, J. K., Why Aren't Operating Systems Getting Faster As Fast as Hardware, *USENIX Summer Conf.*, Anaheim, CA, June 1990, 247-256.

Pai, V. S., Druschel, P., and Zwaenepoel, W., IO-Lite: A Unified I/O Buffering and Caching System, *ACM Transactions on Computer Systems 18*, 1 (Feb. 2000), 37-66.

Palm Inc., PalmOS, (available as http://www.palmos.com), 2000.

Parnas, D., On the Criteria to be used in decomposing systems into modules, *Comm. of the ACM 15*, 2 (1972).

Patterson, D. A. and Ditzel, D. R., The Case for the Reduced Instruction Set Computer, *Computer Architecture News 8*, 6 (Oct. 1980), 25-33.

Pfleeger, C. P., *Security in Computing*, Prentice Hall, Englewood Cliffs, NJ, Second edition, 1996.

Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P., Plan 9 From Bell Labs, *Usenix Computing Systems*, 1995.

Postel, J., Internet Protocol, RFC-791, ISI, Sept. 1981.

Postel, J., Transmission Control Protocol, RFC-793, ISI, Sept. 1981.

Probert, D., Bruno, J. T., and Karaorman, M., SPACE: A New Approach to Operating

System Abstraction, *Proc. of the International Workshop on Object Orientation in Operating Systems*, Palo Alto, CA, Oct. 1991, 133-137.

Probert, D. B., *Efficient Cross-domain Mechanisms for Building Kernel-less Operating Systems*, PhD Thesis, Department of Electrical and Computer Engineering, University of California Santa Barbara, Santa Barbara, CA, Aug. 1996.

Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J., Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures, *IEEE Transactions on Computers 37*, 8 (Aug. 1998), 896-908.

Reiss, S. P., Connecting Tools Using Message Passing in the Field Environment, *IEEE Software 7*, 4 (July 1990), 57-66.

Ritchie, D. M. and Thompson, K., The UNIX Time-Sharing System, *Comm. of the ACM 17*, 7 (July 1974), 365-75.

Rosenblum, M., Herrod, S. A., Witchel, E., and Gupta, A., Fast and Accurate Multiprocessor Simulation: The SimOS Approach, *IEEE Parallel and Distributed Technology 3*, 4 (Fall 1995).

Rosu, M., Schwan, K., and Fujimoto, R., Supporting Parallel Applications on Clusters of Workstations: The Virtual Communication Machine-based architecture, *Cluster Computing 1*, 1 (May 1998), 51-67, Baltzer Science Publishers.

Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Leonard, P., Langlois, S., and Neuhauser, W., Chorus Distribted Operating System, *USENIX Computing Systems 1* (Oct. 1988), 305-379.

Saulpaugh, T. and Mirho, C. A., *The Inside JavaOS Operating System*, Addison Wesley, Reading, MA, 1999.

Schröder-Preikschat, W., *The Logical Design of Parallel Operating Systems*, Prentice Hall, Englewood Cliffs, NJ, 1994.

Seawright, L. H. and Mackinnon, R. A., VM/370 – A Study of Multiplicity and Usefulness, *IBM Systems Journal 18* (1979), 4-17.

Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A., Dealing With Disaster: Surviving Misbehaved Kernel Extensions, *Proc. of the Second USENIX Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996, 213-227.

Shapiro, J. S. and Weber, S., Verifying the EROS Confinement Mechanism, *2000 IEEE Symp. on Security and Privacy*, Berkeley, CA, May 2000, 166-176.

Shapiro, M., Structure and Encapsulation in Distributed Systems: the Proxy Principle, *Proc. of the Sixth IEEE Symp. on Distributed Computer Systems*, Cambridge, MA, May 1986, 198-204.

Shapiro, J. S., Farber, D. J., and Smith, J. M., The Measured Performance of a Fast Local IPC, *Proc. of the Fifth International Workshop on Object Orientation in Operating Systems*, Seattle, WA, Oct. 1996, 89-94.

Shapiro, J. S. and Weber, S., Verifying Operating System Security, MS-CIS-97-26, University of Pennsylvania, Philadelphia, PA, July 1997.

Shapiro, J. S., Smith, J. M., and Farber, D. J., EROS: A Fast Capability System, *Proc. of the 17th Symp. on Operating System Principles*, Kiawah Island Resort, SC, Dec. 1999, 170-185.

Shock, J. F., Dalal, Y. K., Redell, D. D., and Crane, R. C., Evolution of the Ethernet Local Computer Network, *IEEE Computer 15* (Aug. 1982), 10-27.

Sirer, E. G., Security Flaws in Java Implementations, (available as http://kimera.cs.washington.edu/flaws/index.html), 1997.

Smith, S. W. and Weingart, S. H., Building a High-Performance, Programmable Secure Coprocessor, *The International Journal of Computer and Telecommunications Networking 31* (1999), 831-860, Elsevier.

Snyder, L., On the Synthesis and Analysis of Protection Systems, *Proc. of the Sixth Symp. on Operating System Principles*, Nov. 1977, 141-150.

Soltis, F. G., *Inside the AS/400: Featuring the AS/400e Series*, Duke University Press, Second edition, 1997.

Spector, A. Z., Performing Remote Operations Efficiently on a Local Computer Network, *Comm. of the ACM*, Apr. 1982, 246-260.

Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Anderson, D., and Lepreau, J., The Flask Security Architecture: System Support for Diverse Security Policies, *The Eight USENIX Security Symp.*, Washington, DC, Aug. 1999, 123-139.

Steele, G. L., Multiprocessing Compactifying Garbage Collection, *Comm. of the ACM 18*, 9 (Sept. 1975), 495-508.

Stevens, W. R., *TCP/IP Illustrated, Volume 1: The protocols*, Addison Wesley, Reading, MA, 1994.

Stevenson, J. M. and Julin, D. P., Mach-US: UNIX On Generic OS Object Servers, *USENIX Conf. Proc.*, New Orleans, LA, Jan. 16-20, 1995, 119-130.

Stodolsky, D., Chen, J. B., and Bershad, B. N., Fast Interrupt Priority Management in Operating System Kernels, *USENIX Micro-kernel Workshop*, San Diego, CA, Sept. 1993, 105-110.

Stroustrup, B., *The C++ Programming Language*, Addison Wesley, Reading, MA, 1987.

Sullivan, K. and Notkin, D., Reconciling Environment Integration and Software Evolution, *ACM Transactions on Software Engineering and Methodology 1*, 3 (July 1992), 229-268.

SunSoft, Java Servlet Development Kit, (available as http://java.sun.com/products/servlet/index.html), 1999.

Sun Microsystems Inc., *The SPARC Architecture Manual*, Prentice Hall, Englewood Cliffs, NJ, 1992.

Sunderam, V. S., PVM: A Framework for Parallel Distributed Computing, *Concurrency: Practice & Experience 2*, 4 (Dec. 1990), 315-339.

Tanenbaum, A. S., Mullender, S. J., and Renesse, R., Using Sparse Capabilities in a

Distributed Operating System, *The Sixth International Conf. in Distributed Computing Systems*, Cambridge, MA, June 1986, 558-563.

Tanenbaum, A. S., *Computer Networks*, Prentice Hall, Englewood Cliffs, NJ, Second edition, 1988.

Tanenbaum, A. S., Kaashoek, M. F., Van Renesse, R., and Bal, H. E., The Amoeba Distributed Operating System - a Status Report, *Computer communications 14*, 6 (July 1991), 324-335.

Tanenbaum, A. S. and Woodhull, A. S., *Operating Systems: Design and Implementation*, Prentice Hall, Englewood Cliffs, NJ, Second edition, 1997.

Tao Systems, Elate Fact Sheet, (available as http://www.tao.co.uk/2/tao/elate/elatefact.pdf), 2000.

Teitelman, W., A tour through Cedar, *IEEE Software 1*, 2 (1984), 44-73.

Thacker, C. P., Stewart, L. C., and Jr., E. H. S., Firefly: A Multiprocessor Workstation, *IEEE Transactions on Computers 37*, 8 (Aug. 1988), 909-920.

Thitikamol, K. and Keleher, P., Thread Migration and Communication Minimization in DSM Systems, *Proc. of the IEEE 87*, 3 (Mar. 1999), 487-497.

Transvirtual Technologies Inc., Kaffe OpenVM, (available as http://www.transvirtual.com), 1998.

UK ITSEC, UK ITSEC Documentation, (available as http://www.itsec.gov.uk/docs/formal.htm), 2000.

Ungar, D. and Smith, R. B., Self: The Power of Simplicity, *Proc. of Object-Oriented Programming Systems, Languages and Applications, ACM SIGPLAN Notices*, Orlando, FL, Oct. 1987, 227-242.

Vahalla, U., *UNIX Internals: The New Frontiers*, Prentice Hall, Englewood Cliffs, NJ, 1996.

Van Doorn, L., A Secure Java Virtual Machine, *Proc. of the Ninth Usenix Security Symp.*, Denver, CO, Aug. 2000, 19-34.

Van Doorn, L. and Tanenbaum, A. S., Using Active Messages to Support Shared Objects, *Proc. of the Sixth SIGOPS European Workshop*, Wadern, Germany, Sept. 1994, 112-116.

Van Doorn, L., Homburg, P., and Tanenbaum, A. S., Paramecium: An Extensible Object-based Kernel, *Proc. of the Fifth Hot Topics in Operating Systems (HotOS) Workshop*, Orcas Island, WA, May 1995, 86-89.

Van Doorn, L. and Tanenbaum, A. S., FlexRTS: An Extensible Orca Run-time System, *Proc. of the third ASCI Conf.*, Heijen, The Netherlands, May 1997, 111-115.

Van Renesse, R., Tanenbaum, A. S., and Wilschut, A., The Design of a High-Performance File Server., *Proc. of the Ninth International Conf. on Distributed Computing Systems*, 1989, 22-27.

Van Renesse, R., *The Functional Processing Model*, PhD Thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, Holland, Oct. 1989.

Van Steen, M., Homburg, P., Van Doorn, L., Tanenbaum, A. S., and de Jonge, W., Towards Object-based Wide Area Distributed Systems, *Proc. of the International Workshop on Object Orientation in Operating Systems*, Lund, Sweden, Aug. 1995, 224-227.

Van Steen, M., Homburg, P., and Tanenbaum, A. S., Globe: A Wide-Area Distributed System, *Concurrency*, Jan. 1999, 70-78.

Verkaik, P., Globe IDL, Globe Design Note, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, Apr. 1998.

Von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E., Active Messages: A Mechanism for Integrated Communication and Computation, *Proc. of the 19th International Symp. on Computer Architecture*, Gold Coast, Australia, May 1992, 256-266.

Von Eicken, T., Basu, A., Buch, V., and Vogels, W., U-Net: A User-level Network Interface for Parallel and Distributed Computing, *Proc. of the 15th Symp. on Operating System Principles*, Copper Mountain Resort, CO, Dec. 1995, 40-53.

Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L., Efficient Software-based Fault Isolation, *Proc. of the 14th Symp. on Operating System Principles*, Ashville, NC, Dec. 1993, 203-216.

Wallach, D. A., Hsieh, W. C., Johnson, K. L., Kaashoek, M. F., and Weihl, W. E., Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation, *Proc. of the Fifth Symp. on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995, 217-226.

Wallach, D. S., Balfanz, D., Dean, D., and Felten, E. W., Extensible Security Architectures for Java, *Proc. of the 16th Symp. on Operating System Principles*, Saint-Malo, France, Oct. 1997, 116-128.

D. L. Weaver and T. Germond, eds., *The SPARC Architecture Manual, Version 9*, Prentice Hall, Englewood Cliffs, NJ, 1994.

Wegner, P., Dimensions of Object-Based Language Design, *SIGPLAN Notices 23*, 11 (1987), 168-182.

Weiser, M., Demers, A., and Hauser, C., The Portable Common Runtime Approach to Interoperability, *Proc. of the 12th Symp. on Operating System Principles*, Litchfield Park, AZ, Dec. 1989, 114-122.

West, D. B., *Introduction to Graph Theory*, Prentice Hall, Englewood Cliffs, NJ, 1996.

Wetherall, D. and Tennenhouse, D. L., The ACTIVE IP Option, *Proc. of the Seventh SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996, 33-40.

Wilkes, M. V. and Needham, R. M., *The Cambridge CAP Computer and its Operating System*, North Holland, New York, NY, 1979.

Wilson, P. R. and Kakkad, S. V., Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware, *Proc. of the International Workshop on Object Orientation in Operating Systems*, Paris, France, Sept. 1992, 364-377.

Wind River Systems Inc., *VxWorks Scalable Run-time System*, Wind River Systems Inc., 1999.

Wirth, N. and Gütknecht, J., *Project Oberon, The Design of an Operating System and Compiler*, ACM Press, 1992.

Wulf, W. A. and Harbison, S. P., *HYDRA/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.

X.509, ITU-T Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - The Directory: Authentication Framework, June 1997.

Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R., The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, *Proc. of the 11th Symp. on Operating System Principles*, Austin, TX, Nov. 1987, 13-23.

Index

Curriculum Vitae

Name:	Leendert P. van Doorn
Date of birth:	October 17, 1966
Place of birth:	Drachten, The Netherlands
Nationality:	Dutch
Email:	leendert@paramecium.org

Education

Polytechnic	Sept ’86 - Aug ’90	HTS, Haagse Hogeschool, Den Haag, The Netherlands. Bachelor degree.
University	Sept ’90 - May ’93	Vrije Universiteit, Amsterdam, The Netherlands. Master degree.
	Sep ’93 - March ’98	Vrije Universiteit, Amsterdam, The Netherlands. Ph.D. student in Computer Systems. Supervisor: A.S. Tanenbaum.

Work experience

Jan ’90 - Jun ’90: Research Co-op, CWI, Amsterdam, The Netherlands.

Sep ’91 - Sep ’92: Research assistant on the Amoeba project, VU, Amsterdam, The Netherlands.

Sep ’91 - Sep ’93: Teaching assistant for the courses ‘‘Compiler Construction,’’ ‘‘Computer Networks,’’ ‘‘Computer Systems,’’ and ‘‘Programming Languages,’’ VU, Amsterdam, The Netherlands.

Jun ’93 - Sep ’93: Research Intern, Digital Systems Research Center, Palo Alto, CA.

Jun ’94 - Sep ’94: Research Intern, Digital Systems Research Center, Palo Alto, CA.

Jun ’95 - Sep ’95: Research Intern, AT&T Bell Laboratories, Murray Hill, NJ.

April ’98 - present: Research Staff Member, IBM T.J. Watson Research Center, Yorktown, NY.

Publications (refereed)

1. Van Doorn, L., ‘‘A Secure Java Virtual Machine,’’ *Proc. of the Ninth Usenix Security Symposium,* USENIX, Denver, CO, August 2000, 19-34.

2. Caminada, M.W.A., Van der Riet, R.P., Van Zanten, Van Doorn, L., ‘‘Internet Security Incidents, a Survey within Dutch Organisations,’’ *Computers & Security,* Elsevier, Vol. 17, No. 5, 1998, 417-433 (an abbreviated version appeared in ‘‘Internet Security Incidents, a Survey within Dutch Organisations,’’ *Proc. of the AACE WebNet 98 World Conference of the WWW, Internet, and Intranet,* Orlando, FL, November 1998).

3. Van Doorn, L., and Tanenbaum, A.S., ‘‘FlexRTS: An extensible Orca Run-time System,’’ *Proc. of the Third ASCI Conference,* ASCI, Heijen, The Netherlands, May 1997, 111-115.

4. Van Doorn, L., Abadi, M., Burrows, M., and Wobber, E., ‘‘Secure Network Objects’’ *Proc. of the IEEE Security & Privacy Conference,* IEEE, Oakland, CA, May 1996, 211-221 (an extended version of this paper appeared as a book chapter in J. Vitek and P. Jensen (eds.), ‘‘Secure Internet Programming - Security issues for Mobile and Distributed Objects’’, Springer-Verlag, 1999).

5. Van Steen, M, Homburg, P., Van Doorn, L., Tanenbaum, A.S., de Jonge, W., ‘‘Toward Object-based Wide Area Distributed Systems,’’ *Proc. of the International Workshop on Object Orientation in Operating Systems,* IEEE, Lund, Sweden, August 1995, 224-227.

6. Homburg, P., Van Doorn, L., Van Steen, M., and Tanenbaum, A.S., ‘‘An Object Model for Flexible Distributed Systems,’’ *Proc. of the First ASCI Conference,* ASCI, Heijen, The Netherlands, May 1995, 69-78.

7. Van Doorn, L., Homburg, P., and Tanenbaum, A.S., ‘‘Paramecium: An extensible object-based kernel,’’ *Proc. of the Fifth Hot Topics in Operating Systems (HotOS)*

Workshop, IEEE, Orcas Island, WA, May 1995, 86-89.

8. Van Doorn, L., and Tanenbaum, A.S., ‘‘Using Active Messages to Support Shared Objects,’’ *Proc. of the Sixth SIGOPS European Workshop,* ACM SIGOPS, Wadern, Germany, September 1994, 112-116.

Publications (unrefereed)

9. Van Doorn, L. ‘‘Computer Break-ins: A Case Study,’’ *Proc. of the Annual Dutch Unix User Group (NLUUG) Conference,* October 1992, 143-151.