A New Protection Model for Component-Based Operating Systems

Greg Law.

Ph.D. Thesis

City University, School of Informatics

October 11, 2001

Contents

Acknowledgements

The work in this thesis is the result of many discussions with many people. The following peopled (listed in alphabetical order) helped from beta-testing and proof-reading, to bug-fixing and technical suggestions: Karl Brummel, Alan Dearle (Ph.D. examiner), Pete Goodliffe, Ramon van Handel, Paul Kelly (Ph.D. examiner), Sunil Kittur, Patty Kostkova, Julie McCann (Ph.D. supervisor), Peter Osmon, Chris Randall, Julian Smith and Timo Suoranta. Sincere apologies are offered to those who have been omitted — they have the author's assurance it is his forgetfulness and not ingratitude!

Abstract

Protected operating systems multiplex programs onto resources such that they are isolated from one another — that is, concurrently executing programs cannot interfere with each other. A layer of software known as the *kernel* provides this protection to the software layers above it. Untrusted, 'user' programs are prevented from controlling the protection hardware because they are executed when the processor is in *user mode* — a mode of reduced privilege. In user mode, instructions that can be used to circumvent protection are unavailable; the processor's instruction-set is reduced.

This thesis introduces a new operating system protection mechanism termed *SISR* — Software-based Instruction Set Reduction (pronounced *scissor*). Here, all software (including the kernel) executes in the same processor mode, while both language independence and protection are maintained. Untrusted (that is, 'user level') code is prevented from issuing privileged instructions not by reducing the processor's instruction set, but by scanning code prior to its loading; any code found to contain privileged instructions is not loaded. Memory protection is provided through segmentation. SISR leads to improved architectures (that is, simpler and more modular), and improves performance significantly. Its low overheads make fine-grained protection practical, making it especially well-suited to component-based operating systems.

A prototype system has been built for x86-based PCs as a 'proof-of-concept'. Significant improvements in architectures have been delivered. Tasks that have previously been inextricably linked (such as interrupt handling and CPU scheduling) have been separated into distinct components. Experiments have demonstrated significant improvements in performance, compared even to the leanest research operating systems.

Chapter 1

Introduction

This thesis introduces a new operating system protection model called *SISR* (*Software-based Instruction Set Reduction*, pronounced 'scissor'). SISR offers improvements in performance and architectures, and is particularly well suited to component-based operating systems. This chapter identifies the responsibilities of operating systems and protection (detailed more fully in Chapter 3), and emphasizes the short-comings of contemporary systems. The notions of component-based software and component-based operating systems are introduced. The motivation behind this work (detailed more fully in Chapter 2) is also given.

1.1 Operating Systems and Protection

Most operating systems have the ability to multiplex many programs onto one computer so that each program has the illusion that it is running on its own, dedicated system. The most common technique used to create this illusion is to run each program in turn, each executing for some small 'time-slice' (typically a few milliseconds). If the illusion is to be complete, programs running 'concurrently' must be *isolated* from each other such that one program cannot interfere with another. This means that programs that are erroneous (that is, buggy) or malicious (for example, a virus) should not be able to access other programs' resources. These resources include memory, processor time and peripherals. The ability of an operating system to enforce this isolation is known as *protection*. Note the distinction between *protection* and *security*: protection is the mechanism used to isolate programs; security is the policy used to control the protection mechanism (including resource allocation and controlled inter-program communication).

Protection comes at a price: it is expensive in terms of computer resources. In traditional

operating systems (for example, BSD UNIX [84]) a sub-routine call from one program to another takes over 3 orders of magnitude longer than a sub-routine call within the same program [74]. This is because protection requires that 'walls' be erected between programs to isolate them; traversing these walls is expensive.

1.2 Component-Based Software

In modern operating systems, providing the illusion that each program has its own, dedicated computer is less important than the provision of protection. Indeed, dividing work between several programs and having those programs *interact* to achieve some task is seen as useful for several reasons:

- The fault-isolation provided by protection makes systems more robust: failures are limited to the failed program.
- Code reuse is made easier. Different tasks often have a lot in common. If this common work is extracted from programs and placed in a separate program, code duplication can be avoided (programs written solely to support common functionality rather than achieve a specific task are commonly called 'servers').
- Software engineering is improved. Software engineers strive to write *modular* programs, where 'cohesion' is maximised and 'coupling' minimised [107]. High cohesion and low coupling is forced between protected programs.

Each protected program can be thought of as a *component*. Several components are brought together in order to achieve some task. Computing with a few independent and well-defined programs is becoming a thing of the past. With older architectures it is clear with which program a user is interacting at any time. This is not so in modern systems where, for example, a user might be interacting with a Java 'applet' though their web-browser. There the user is interacting with several components: the browser, the Java virtual machine and the applet.

This model can be extended so that the advantages of component-based software engineering are exaggerated. For example, an internet client might be constructed from several components including an 'HTML Renderer', an 'Image Viewer', a 'Movie/Sound Player', a 'Java Virtual Machine', 'SMTP' (e-mail) and 'HTTP' (web) clients, a 'Text Editor' and a 'Spell Checker'. Many of these components can be used in other applications — for example, the spell-checker and

text-editor might be used by a word-processing package, or along with the HTML renderer in a web-authoring package.

1.3 Towards Component-Based Operating Systems

Component-based operating systems embrace component technology in two ways. Firstly, they are designed to support (relatively small) components and their interaction rather than monolithic, single-purpose programs. Traditional systems implement protection between programs where it is assumed that protection boundaries will be crossed rarely. A component-based OS should provide a protection mechanism suitable for finely-grained protection with frequent interaction between protected components. Secondly, the operating system itself is decomposed. That is, rather than providing all OS services from a monolithic entity (often known as the *kernel*), operating system services should be provided by a set of cooperating components.

In some senses, *μ-kernels* (see Section 4.2.2) and even monolithic systems such as UNIX [89] can be seen to be component based. For example, UNIX is comprised of many components, including the kernel, the `login` process, utilities such as `cp` and `ls`, shells and (on later systems) virtual file systems. However, contemporary operating systems cannot be considered truly component based. A component-based operating system should have a component model that is: explicit; consistent and orthogonal; lightweight; and pervasive. A more complete definition of a component-based operating system is given in Section 4.4.

1.4 Motivation

Despite the continuing move towards component-based operating systems, existing systems fall some way short of the two goals of component-based OSs (component support and OS decomposition). In other words, existing systems do not provide as much support for component-based applications as they might, and even in μ-kernel and exokernel-based systems [30], the OS is not as decomposed as it would ideally be. Traditional protection mechanisms have two features that are the limiting factor with respect to both of these short-comings:

- Traditional operating systems' kernels execute while the microprocessor is in a special, privileged mode. This forces all OS services to be bundled together into a single entity — the kernel. For example, even the smallest kernels have support for many disparate services

including interrupt handling, paging, preemptive scheduling, memory management and resource allocation and ownership tracking. In a decomposed system, separate components should be responsible for these distinct tasks. Such decomposition would lead to improved configurability, dynamism, robustness and software engineering.

- Even in the leanest research systems (such as L4 [45]), protection is still so expensive as to prohibit protection at a particularly fine granularity (see Section 2.3.2). This means that if component-based operating systems' performance is to remain acceptable then either decomposition must be limited, or several components must be placed into a single protection context.

This thesis presents a new protection mechanism, *Software-based Instruction Set Reduction.* SISR solves both the above issues at the expense of rarely-used and discouraged techniques such as self-modifying code[1]. Traditional systems have untrusted code execute while the processor is in a special mode of reduced privilege. This mode renders privileged instructions (usually those instructions that control protection) unusable. SISR is a simple technique whereby untrusted code is scanned prior to execution to ensure that no privileged instructions are present. All code then executes while the processor is in its most privileged mode. Furthermore, the code-scanning means that user-level components' use of certain instructions (such as segment-register loads) can be restricted by the operating system. By combining code-scanning with memory protection based on segmentation, SISR allows better decomposition of the kernel (since it is no longer 'special'), as well as significantly reducing the costs of traversing protection contexts.

Over the years, OSs have had smaller and smaller kernels, from monolithic, through μ-kernels and to 'nano-kernels' such as exokernels. A truly component-based OS can be seen as a "zero-kernel", where the kernel has been replaced by a set of components that cooperate to provide services usually found in the kernel. SISR allows the construction of such zero-kernels through the abolition of separate processor modes and the dramatically decreased costs of decomposition.

1.5 Structure of this Thesis

The remainder of this thesis is structured as follows: Chapter 2 specifies the deliverables of this work — explicit requirements are identified, on which the success of the project may be judged.

[1] "Self modifying code" refers to code that writes to itself during execution. This should not be confused with dynamic code generation techniques where some server produces or modifies a client's code prior to the client's execution. That is, the term self-modifying code does *not* refer to modern techniques such as Aspect Oriented Weaving [75] or Java Just-In-Time compilation [23].

Chapters 3 and 4 set the context for the rest of this thesis: Chapter 3 gives an overview of the protection mechanisms used by existing, main-stream operating systems; Chapter 4 introduces research related to that presented here (namely, significant research into new protection mechanisms). Chapter 5 presents the SISR protection model in detail, and explains how the goals outlined in Chapter 2 will be met. Chapter 6 gives an overview of an operating system constructed using SISR as a 'proof-of-concept'. Chapter 7 presents several experiments conducted on the new operating system and their results. Chapter 8 concludes against the goals set out in Chapter 2 and suggest directions for future work with SISR.

Chapter 2

Project Goals

2.1 Introduction

This chapter introduces current OS research topics, and goes on to specify those to be addressed in this work. Briefly, these are: software engineering, performance, configurability (also known as composability) and dynamism (also known as reconfigurability). Furthermore, these goals must be met while maintaining language independence (that is, features of a programming language cannot be used to achieve these goals, since that would restrict software engineering). The chapter also explains how many of the goals are interdependent.

2.2 Current OS Research Goals

There are several 'hot topics' in operating systems research. These are:

Performance Moore's law states that μ-processor performance doubles approximately every 18 months. However, the same cannot be said of operating system performance [86]. For example, the performance of 'number-crunching' or graphics manipulation is improving rapidly, whereas the time taken to perform context-switches has remained largely static. Recent projects such as L4 have striven to redress the balance by constructing operating systems with the intent of allowing the OS (and thus applications) to get the most out of modern μ-processors.

Configurability Different operating system services are required in different domains. A configurable operating system is able to assume many guises, and so allow different configurations

of the same operating system to be used in different domains. For example, a configurable operating system might have a general-purpose preemptive scheduler in desktop distributions, and a real-time scheduler in embedded ones.

Flexibility Not only are different operating system services required in different domains, but different *applications* often require different services of the same operating system. Furthermore, these different applications may well be executing concurrently. Systems such as exokernels [30] and Cache-Kernels [18] allow different applications to see different 'views' of the same OS, depending on their needs. Applications are free to *customise* the operating system, so that the OS may exhibit the behaviour most suitable to them. Note that configurability refers to OS customisation by the vendor or customer, whereas flexibility refers to OS customisation by the application designer.

Dynamism Most configurability and flexibility offered by today's commodity operating systems involves the system being re-booted between changes. However, this is not acceptable where systems must be highly available (such as corporate servers or industrial control systems). Allowing operating systems to change their behaviour dynamically would bring the benefits of configurability and flexibility to such systems. It will also allow bug-fixing of systems software without requiring a suspension of service. Dynamism is also referred to as hot-swapping and *re*configurability.

Software-engineering Systems programming is notoriously difficult and error-prone, yet relatively little effort is focused on easing the task of the systems developer. Reducing the cost of systems-software development will allow more individuals and companies to engage in such development.

Security As the world becomes ever-more reliant on computing, security becomes an increasingly important issue. Current commodity operating systems offer relatively poor security, with all systems having a plethora of well-documented 'security holes' that can be exploited to break security.

While distinct, many of these goals are interdependent. Flexibility, configurability and dynamism are much more useful if application developers are able to customise the system themselves; something not practical until significant advances are made in systems-software engineering. Exokernels and Cache-kernels have demonstrated that flexibility is key to offering improved performance. Flexibility and configurability are obviously related since they both involve customisation

of operating system services. Furthermore, all goals can be argued to be dependent on good performance being maintained. That is, history has show that no matter what advantages an operating system offers, no OS will be accepted if it performs poorly. For example, for many years MS-DOS and Windows 3.1 remained more popular than Windows NT or UNIX, despite Windows 3.1's notorious instability. Stable systems became popular only once cheap, commodity hardware was capable of running OSs such as NT and UNIX responsively.

2.3 Project Goals

This thesis describes a new OS protection model which is intended to provide significant improvements in many of the requirements outlined in Section 2.2. Namely: performance; configurability; dynamism and software-engineering.

This section examines each goal in detail, laying down criteria on which the success of the project may be judged. Each of the goals mentioned here must be met for the work to be judged a complete success. As noted in Section 2.2, many operating system research goals are interdependent. For example, the fine granularity necessary to meet the software-engineering requirements is dependent upon high performance — if protection is too expensive modules that would ideally be protected from one another end up being collocated into a single protection-context [21].

2.3.1 Software-engineering

Software-engineering will be improved through the use of a consistent component model that is pervasive throughout the system. These components are to be protected *and* finely-grained. As well as leading to improved system modularity, the increased granularity of fault isolation will ease debugging [65]. The kernel itself will be decomposed, easing the burden of systems-software engineering. In other words, the system will exhibit the four key properties of a component-based operating system identified in Section 1.3: orthogonality, consistency, pervasiveness and lightweightedness.

Furthermore, [99] shows that the majority of errors in systems-programming are invalid memory references. A protection model that is lightweight and flexible enough to allow fine-grained protection at the programmer's abstraction boundary will improve systems-software engineering significantly.

Since poorly-performing systems tend to be unpopular, this finely-grained decomposition must

not adversely effect performance. Hence this requirement is dependent on the performance requirements (Section 2.3.2).

Software-exception support will also be included to aid software-engineering [91, 19].

2.3.2 Performance

The new protection model presented here should reduce protection overheads to enable fine-grained protection with a minimal performance hit. Traditionally, when using a component architecture such as CORBA [41], the high overheads of protection force the programmer to compromise protection by using collocation (that is, objects that would ideally be protected from one another are grouped together for performance reasons). The new protection model should allow protection of 'finely grained' systems with less than 10% spatial and temporal overheads.

Temporal Overheads

Traditional context-switching is slow. The penalty imposed by crossing context boundaries has previously acted as a deterrent to extensive decomposition of software. Java security-boundary crossings have been measured at 30,000 per second on a 167MHz Ultra-SPARC machine [109]. Rounding this figure up to 100,000 boundary crossings per second, this translates as an invocation approximately once every 1,500 cycles. Therefore, if protected method invocations are to impose no more than 10% overhead, **a round-trip RPC must take no more than 150 cycles**.

Note that Remote Procedure Call (RPC) is by far the most popular software-engineering tool for cross-protection-domain communication, hence the inter-component communication overheads requirements apply to *RPC overheads*, not some other primitive such as message-delivery times.

Spatial Overheads

The space overhead per component must remain low. Traditional, page-based memory protection often requires several pages per protection-context. As an example, Linux [11] requires at least 6 pages per protection domain[1] [57]. Assuming even distribution of protection-context sizes, $\frac{1}{2}$ a page will be lost to internal fragmentation for each page used. Since Linux uses 6 pages per process, this means that $3nP$ bytes are lost to internal fragmentation (where P is the page size, and n the number of processes). In a UNIX process model with a few dozen processes running at any one time, this overhead is acceptable since memory is relatively cheap. However, in finely-grained

[1] Linux requires at least 1 page for user-mode stack and data; at least 1 for user-mode code; 1 for a kernel-mode stack; 1 for a 'Task-State Structure'; 1 for the process's page directory; and at least 1 for the process's page tables.

component-based system, n is likely to be very large (at least several thousand). Furthermore, the phrase 'memory is cheap' applies only to desktop systems. Embedded systems typically have relatively tight memory requirements because memory is not *free*, and when manufacturing consumable devices in large quantities "every penny counts". Secondly, 'cheap' memory refers to DRAM, which has a significant power consumption that scales linearly with the memory size.

It is anticipated that the granularity of protection will be fine, with the average component size being perhaps as low as 1kB. This means that the protection overheads must be less than 100 bytes per protection domain if fine-grained protection is to impose no more than a 10% overhead.

2.3.3 Configurability

An operating system built of finely-grained components is likely to provide high configurability. The configurability of most systems is hindered because programmers employ 'hacks' that bypass interfaces, introducing strong coupling between modules (particularly as software evolves). Since the protection of components enforces encapsulation, protected components are guaranteed to communicate only via their interfaces. Thus it is argued here that any system comprising of protected components will naturally offer improved configurability.

2.3.4 Dynamism and Availability

Many current operating systems require a re-boot after re-configuration before changes take effect. This situation is improving slowly, and many modern systems allow device drivers to be installed with no interruption of service. However, more fundamental changes, such as the installation of a new kernel, still require a reboot. This is not acceptable for systems that are required to be available permanently. The component-model used should support 'hot-swapping' of components. This means that the implementation of a component may be swapped with another realising the same interface, without clients of that component being aware.

How exactly state is transferred between the incoming and outgoing implementations is not to be addressed here — this is left to the application programmer. Although it could be argued that transfer of state is the most difficult part of hot-swapping, the protection model itself is of no relevance here, and so state transfer between versions is beyond the scope of this work. Moreover, subsequent versions introducing small changes such as bug-fixes will often require no manipulation of a component's state – that is, the new version will be able to carry on with the old one's state. In the case that more complicated state manipulation is required, this can be

handled on a case-by-base basis, or by building some more complicated scheme on top of the new protection model.

2.3.5 Language Independence

If an operating system is to be widely accepted it is important that it is language independent. This means language features such as type safety cannot be relied upon to achieve any of the goals outlined in this section.

2.4 Research Goals Specifically *Not* Targeted

There are several areas of operating systems research that this work does not attempt to address (although it must not result in a system where these areas are any weaker than in commodity operating systems). The most striking goals common in modern research OSs not covered here are flexibility and security.

2.4.1 Flexibility

While flexibility is a 'hot topic', it requires a 'flexible architecture' be implemented on top of the protection model. That is, an operating system's flexibility is not a feature of its protection mechanism. Because this thesis introduces a new protection model, *improvement in flexibility is not a goal of this work.* However, the new protection model presented here should in no way hamper flexibility. That is, it should be at least as easy to build a highly flexible system using the new protection model as on other protection models such as exokernels.

2.4.2 Security

As described in Section 3.6, security and protection can be viewed as orthogonal issues. The protection model presented here will allow systems to be at least as secure as commodity systems such as UNIX [89] to be built on top. However, this work does not aim to provide any unusual levels of security, or any novel security mechanisms.

2.5 Summary

This chapter has identified the active OS research goals, and the criteria by which success or failure of this work may be judged. The key motivation behind the project is to provide improved software-engineering, mainly through lightweight, but protected components. Specifically, protected round-trip null-RPC must take less than 150 cycles, and the spatial overhead of a protected component must be not more than 100 bytes. Furthermore, the resulting system must be language-independent, and software-exception support should be included. Decomposition should result in improved configurability. Allowing components to be hot-swapped will allow improved dynamism and availability. *All* of these goals will be met by the new protection model, developed throughout the remainder of this thesis.

Chapter 3

An Overview of Operating Systems and Protection

3.1 Introduction

This chapter sets the context for the work presented in the remainder of this thesis. That is, before a new operating system protection paradigm is presented, it is necessary to introduce the notions of an *operating system* (OS) and *protection*. The chapter then goes on to describe the mechanisms usually employed to implement protection, in particular, the hardware and abstractions used by most systems are examined. The terms 'Trusted Computing Base' and 'OS kernel' are also defined. The notion of 'security' is also examined, as is how security is related to (but is different from) protection.

3.2 Operating System Responsibilities

Although most computer-literate people understand what is meant by the term 'operating system' there is much confusion about what exactly comprises an OS. Essentially, an operating system can be defined as a layer of software that sits between application programs and the computer hardware on which they run. However, the responsibilities of different operating systems vary: some OSs abstract the hardware, presenting an environment more friendly to applications (for example, MS-DOS [100]), others multiplex several applications onto the hardware concurrently so that different application may not interfere with one another (for example, Exokernel [30]), and

others do both (for example, UNIX [89]).

This thesis concentrates on the role of the operating system to support several applications to be run concurrently on one computer but remain isolated from one another. This feature is known as *protection*. Specifically, protection refers to the division of a computer's resources between concurrently executing applications so that an application may not use any of another's resources. A protected operating system ensures the integrity of the whole system even in the presence of malicious applications (for example, a virus) as well as legitimate applications that contain programming errors.

Protection is mostly concerned with the two major hardware components of any computer system: the central processing unit (CPU) and memory. Although most protected operating systems also protect peripherals (such as disk drives, network bandwidth, and visual display units), this thesis concentrates on the protection of memory and CPU time. This is because the overheads of protection are traditionally very high relative to the fast CPU and memory, but insignificant for the much slower peripherals. Also, most modern computer systems employ 'memory-mapped' peripherals — that is, peripherals are control by accessing certain 'special' memory locations. This means that once memory protection is enforced, peripheral protection follows naturally.

Operating system *protection* is often confused with OS *security*. While security is related to protection, it is a quite different topic. The term 'security' refers to the policy used to determine exactly what resources are given to an application. The protection mechanism restricts applications so that they may access only the resources allocated to them by the security policy. A brief overview of security issues is presented in Section 3.6.

3.3 Protection Implementation

While the security policies of different OSs are often very different, most operating systems implement protection in the same way. Protection is traditionally implemented using a combination of hardware and software: the hardware architecture is organised so as to make protection possible, but software is needed to control the protection hardware properly. This software is commonly known as the OS 'kernel'. This section describes how most OS kernels drive the hardware to provide protection. Briefly, there are 3 facets to protection:

Memory Protection Preventing applications accessing memory not allocated to them

CPU Protection Preventing applications from monopolising the CPU

Privileged Instruction Protection Preventing applications from executing instructions that can be used to compromise protection (this can be thought of as 'meta-protection'.

The following sections describe these three aspects of protection and the implementation in more detail. Further details of common memory-protection techniques are given in Section 3.4.

3.3.1 Memory Protection

In order to prevent one program from accessing the memory of another it is common to confine each program to its own *protection domain* (sometimes known as a *process*[1]). A protection domain is the memory to which the associated program has exclusive access.

The most common technique used to effect memory protection uses dedicated memory-management hardware that limits the memory ranges accessible (see Section 3.4 for details). The kernel can manipulate this memory-management hardware in order to make different programs' protection domains accessible at different times. By coinciding this with the scheduling of programs on a CPU, each program is limited to its own protection domain, and memory protection is achieved.

Untrusted software is prevented from manipulating the memory protection hardware by two means. Firstly, instructions that manipulate the memory hardware are privileged so that user-mode programs may not execute them (see Section 3.3.3). Secondly, the memory protection hardware relies on certain data-structures. These structures are setup so that they themselves are inaccessible from all software other than the kernel — that is, they reside in the kernel's protection domain.

One protection domain is active on each CPU at one time. The entire processor's state (including the active protection domain) is often known as a *context*. In older operating systems, a context usually had associated with it a single thread of control, which included its own 'register context' (that is, the state of CPU registers). As contexts are switched between very frequently, the illusion of one computer per context is created, even though several contexts are multiplexed onto a single CPU and memory system. In more recent operating systems, each context can usually have more than one thread. Each thread is associated with one protection domain at a time (that is, the memory that thread can access), but some systems allow threads to change the domain with which they're associated (see Section 3.5 for more details).

[1]The term process is sometimes used to describe a domain where both memory and CPU are protected (that is, a complete program in execution), and sometimes refers just to a memory-protection context.

3.3.2 CPU Protection

In order to ensure fair allocation of CPU time between threads, it is most common to employ *interrupts*. This is a hardware mechanism used to force control from the currently executing application to a well-defined instruction within the kernel. Interrupts are triggered by peripherals attached to the processor. A timer device usually triggers an interrupt periodically, in response to which the kernel either returns control to the interrupted program, or schedules another. When scheduling another application the current application's protection domain must be swapped for the new application's (known as a 'context switch'). By context switching on periodic interrupts like this, the kernel can ensure that no program is able to monopolise the CPU. Some systems (known as real-time systems) guarantee that a program will receive a certain amount of resource so that tasks may be performed within strict time constraints (for example, an aircraft control system will need to guarantee that a control-surface responds to commands within a given number of milliseconds if stable flight is to be assured). However, real-timeliness usually has associated costs: systems that are not real time (known as desktop or general-purpose systems) can usually achieve a higher *throughput* (that is, perform more work per second), at the cost of predictability.

Faults are similar to interrupts, but are triggered internally rather than in response to peripherals (usually when the processor cannot complete the current operation). For example, if a divide instruction is issued with a divisor of zero, the processor will trigger a fault, forcing control to a pre-determined offset within the kernel. Similarly, attempts to access memory not part of the current protection domain result in fault. On receiving faults, the default action of the kernel is usually to terminate the faulting application. However, depending on the nature of the fault, the kernel might remedy the cause of the fault and retry the faulting instruction; synthesise the faulting instruction and resume execution after the faulting instruction; signal the fault to the application; or take some other action.

In summary, interrupts are triggered by peripherals and faults are triggered by executing some invalid instruction, but both use the same mechanism.

3.3.3 Privileged-Instruction Restriction

Most processors' instruction sets are defined so that most instructions can affect only the program that issues them. The processor can be placed into a mode of reduced privilege where only these instructions may be executed (known as *user mode*). All applications execute while the CPU is in 'user mode', and the kernel executes while the CPU is in 'kernel mode'. Indeed, the kernel can

be defined as the code that executes while the processor is in kernel mode.

The few instructions that, if misused, can cause the system as a whole to fail are known as *privileged instructions*. The attempted execution of a privileged instruction while the processor is in user mode causes control to transfer to the kernel via a *fault*.

Interrupts and faults place the CPU in kernel mode, regardless of the processor's mode before they are triggered. When applications require access to the kernel (in order to request some services) they can issue a fault. Most processors provide an explicit instruction that will 'fault' — issuing such an instruction is known as a *system call*. Note that because the processor is placed into kernel mode and transfer controlled to the kernel *atomically*, protection is maintained (that is, rogue applications cannot place the CPU in kernel mode and retain control). Because system calls use the same mechanism as interrupts, they are sometimes called 'software interrupts'.

3.4 Hardware Memory-Protection Mechanisms

Over the years, different mechanisms have been used to implement memory protection in hardware. The following sections explore the most common variants: paging, segmentation and a combination of the two. Each type of memory-management hardware described here allows memory to be addressed by a *virtual* rather than the *physical* address. That is, the memory addresses generated by the executing program are known as virtual addresses. These are not fed directly to the memory system. Instead, addresses are *translated* by memory-management hardware before being used to look up values in the memory system (the address used to index the memory system is known as the *physical* address). The hardware that performs this translation is usually located "on chip" and known as the Memory-Management Unit (MMU).

3.4.1 Paging

The most popular form of memory protection is *paging*. Here, memory is divided into uniformly sized *pages* (each page is typically a few kilobytes). Each page has associated with it a set of *access permissions*, and a *translation*. The access permissions and translations for each page are stored in a structure known as the *page-table*. Before accessing a given memory location, the memory subsystem checks against the permissions of that page, and performs the translation as given in that page's entry in the page-table. This is shown in Figure 3.1.

In Figure 3.1, a 4kB page-size in a 32 bit system is assumed (that is, a 12 bit page offset leaving

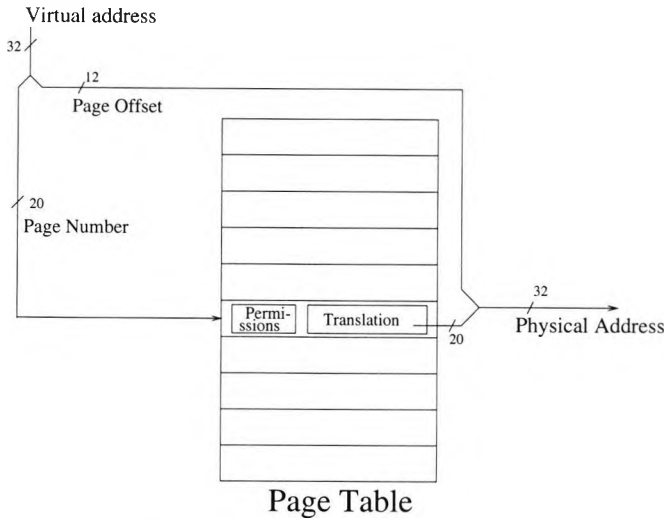


Figure 3.1: Memory protection with paging

a 20 bit page number).

Since a 20 bit page number requires 2^{20} page-table entries, and because page-tables are usually sparsely populated (that is, most machines have much less than the maximum amount of memory physically installed), a one-dimensional linear table would usually be far too big. In reality, various techniques such as two-level tables, inverted tables or hash-tables are used to reduce the page table's size. However, requiring several memory accesses to find the translation and access rights per memory reference is prohibitively expensive, particularly since memory latency is usually a significant bottleneck on a computer's throughput. To avoid this penalty on each access, a Translation Look-aside Buffer (TLB) is typically used to store a cache of translations and access rights "on-chip". Traditionally, the TLB cache is managed by the MMU. Some recent systems (such as the DEC Alpha [22]) require management of the TLB cache in software (via 'TLB-miss faults'), allowing the operating system rather than the hardware to define the structure of the page-table, and TLB replacement policies.

Each protection domain usually has its own page table associated with it. In order to perform a context switch the outgoing domain's page table is replaced by the incoming domain's (usually by loading the physical address of the new page table into some register on the MMU). If a TLB is employed, that must also be updated so that it reflects the new page-table.

Pages' *translations* are often used to implement *demand-paged virtual memory*: giving the illusion that a machine has more physical memory than is actually installed. When physical memory is exhausted, the contents of infrequently used pages can be stored out on some secondary

storage (usually disk), and the physical memory backing these pages can be reclaimed. This process is known as "swapping a page out to disk". The access permissions of pages stored on disk are marked so that the page is not accessible at all. Any subsequent attempt to read or write a swapped page will trigger a page-fault. The operating system responds to such faults by allocating some physical memory and copying the page's contents in from secondary storage (known as "swapping in"). Note that allocating physical memory used to swap in a page often requires that some other page is swapped out to disk. Confusingly, the term virtual memory is often used to refer to this technique. For clarity, this technique will be described as "demand-paged virtual memory" in this thesis, and the term "virtual memory" will be used to describe hardware-based memory protection in general.

Pages' translations are also used to give each protection domain its address space. For example, two protection domains might both grow their address-space during execution. Both domains can preserve a contiguous virtual address range, even if the physical memory of both address spaces is interleaved. The OS designer has a choice of whether to allow different protection domains' virtual address spaces to overlap. Allowing virtual address spaces to overlap relieves the contention on virtual address ranges, but complicates shared memory. Traditionally OSs have allowed the virtual address ranges of protection domains to overlap. However, modern 64-bit architectures offer huge[2] address spaces where virtual address ranges are no longer a contended resource. Many 64-bit operating systems do not allow domains' address spaces to overlap, allowing pointers to be meaningful across protection domains. Such systems are commonly referred to as single address-space operating systems, or SASOSs [14, 15, 16].

Paging was developed as part of the University of Manchester's MU5 computer [50], and is supported by virtually all modern processor architectures.

3.4.2 Segmentation

Though not as popular as paging, segmentation is another significant hardware-based memory protection technique. There are several versions of segmentation: the one shown here is that found on the Intel IA32 architecture [52]. Other architectures present segmentation in slightly different ways, although all common models share the same key principles. Like a page, a *segment* is a region of memory with a translation and access rights. However, unlike pages, segments have

[2] 2^{64} is a bigger number than there are millimetres in a light-year!

a variable size, usually with byte or word granularity[3]. The processor also maintains a set of currently 'active' segments, with different segments for (say) code, data and stack — all code is loaded via the code segment, data accesses go via the data segment, and stack operations go via the stack segment. Segments are described by a *descriptor* (analogous to a page-table entry on paged systems), which contains (among other attributes) the segment's base in physical memory, its size, and its access permissions. The set of segments in a system is described by a *descriptor-table* (analogous to a page-table). An individual segment is identified by its index into this table, known as a *selector*.

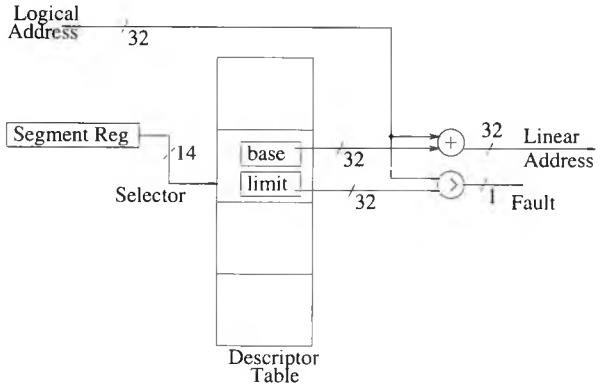


Figure 3.2: Memory protection with segmentation

Figure 3.2 demonstrates such segmentation, assuming a selector size of 14 bits, and a machine-word width of 32 bits. Note that the permission check is performed as part of address translation.

As with paging, context switches can be effected by changing the descriptor-table so that outgoing segments are inaccessible, and incoming ones accessible.

Segmentation has the advantage over paging that segments are variable-sized, meaning one segment can fit any protected memory region exactly. This means that fewer segments than pages are required, reducing memory-management overheads, and also that memory is not lost to *internal fragmentation*. Internal fragmentation describes the situation where more memory is allocated than used — protecting a 1 byte region of memory with a 4-kB page size would consume 4kB, meaning that 4095 bytes are lost to internal fragmentation. However, because segments are contiguous and variable-size, segmented systems are prone to *external* fragmentation. External fragmentation is used to describe the situation where memory is unused but cannot be allocated (for example, although there may be sufficient free memory in total to satisfy an allocation request,

[3]Unlike pages, segments do not have a 'typical size'. Segments may have a size between a few bytes up to several gigabytes, depending on their use.

the request fails because there is no single contiguous block of free memory large enough). Also, the addition required by the segmentation hardware for each memory access is considerably more work than the concatenation required by paging, potentially hampering a segmented architecture's performance[4]. However, virtual caches [46] (also known as virtually indexed caches) can be used to avoid any overhead in the common case.

Though not as popular as paging, segmentation is a viable protection technique. The MULTICS system [33], IA32 [52], the POWER [3] and PA-RISC [59] architectures all use segmentation (although not necessarily in exactly the form described above).

3.4.3 Orthogonal Segmentation and Paging

The Monads [61, 60] project resulted in a new memory-protection model whereby paging and segmentation were combined *orthogonally*. Here, segmentation is used as normal to translate the logical addresses. However, the resultant address (known as the *linear* address) is then passed through page-tables to form the physical address. This is shown in Figure 3.3.

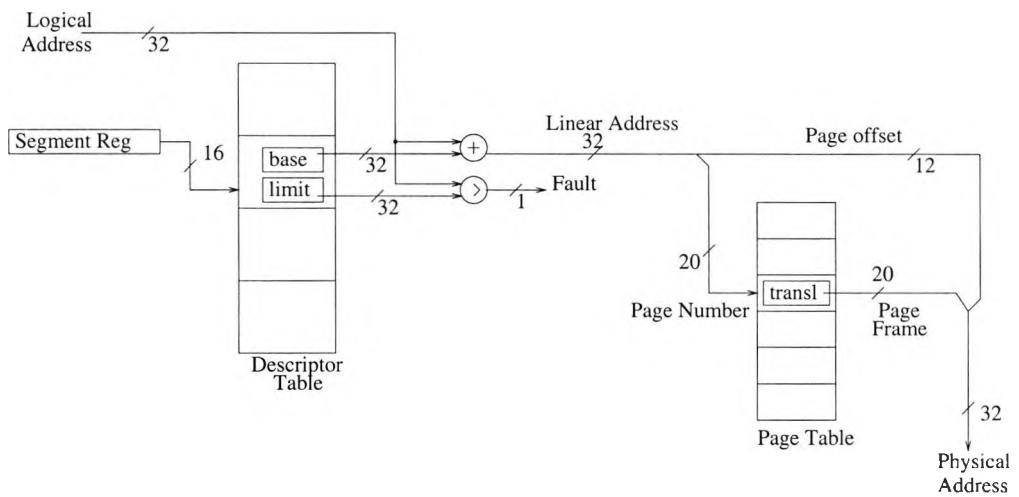


Figure 3.3: Memory protection with orthogonal segmentation and paging

This scheme combines the main advantages of both paging and segmentation — most notably provision for the elimination of internal *and* external fragmentation.

Notably, IA32 uses this scheme of memory protection, although most operating systems employ the so called 'flat-model' whereby 1 segment maps the entire linear protection domain[5] and all protection is realised through paging. This is not to be confused with the Intel 8086 version of

[4] Naively implemented segmentation results in architectures with either longer cycles, or extra pipeline stages.

[5] In reality, protected operating systems on IA32 require 4 segments: 1 data and 1 code segment for both kernel and user privilege-levels.

'segmentation' which is not the segmentation method described, but a 'hack' to allow a 16 bit processor to address 1MB of memory. The Intel versions of L4 [45] and QNX [47] make use of both paging and segmentation to improve context-switch times while providing applications with a conventional, UNIX-like environment.

3.5 Cross-Domain Communication (IPC)

If several protection domains are to cooperate in order to perform useful work, then these domains must be able to communicate in a protected manner. There are several techniques used to allow communication between different protection domains. This section will examine the most common techniques. Cross-domain communication is often referred to as Inter-Process Communication or IPC (somewhat confusingly, the term IPC is often used even when a protection domain is not termed a process!).

Shared Memory. Most operating systems allow two or more domains to share a subset of their protection domain. This shared memory can be used to effect communication between domains. Shared memory is very flexible, but this flexibility can be considered a cost as well as a benefit — the lack of enforced semantics can complicate application design significantly.

Message-Passing. Probably the most common technique used to communicate between protected protection domains is *message-passing*. Here, a protection domain sends or receives encapsulated messages to or from another. Primitives are usually available that send a message and block on its reply atomically in order to optimise synchronous communication. Some systems allow (or require) that *channels* (also known as *sockets*) be bound between the communicating domains over which messages are subsequently transmitted.

Messages can be passed by explicit copying, or by re-mapping pages or segments from the sender to the receiver's domain. This is useful for the transmission of large messages, providing effectively infinite bandwidth. However, this high bandwidth is provided at the expense of latency, and so is inappropriate for short messages.

Pipes/FIFOs. Pioneered in UNIX, pipes [62] allow one domain to transmit characters to another as though it were writing to a file. Similarly, characters can be received as though reading from a file. While a useful abstraction, pipes still require some communications primitive underneath. Indeed, a pipe can be thought of simply as a channel for the transmission of single-byte messages.

Thread-Tunnelling. Many recent operating systems have exploited thread-tunnelling (also known as thread migration or the passive object-model) [35]. Thread-tunnelling is functionally equivalent to synchronous message passing, except that rather than sending a self-contained message from the client to the server, the client thread *migrates* to the server's protection domain. In such systems, threads of control are not statically bound to protection domains: a domain might have zero, one or many threads of control associated with it at any moment.

3.5.1 Comparison

Message-passing and thread-tunnelling can be considered functionally equivalent, particularly when one considers message-sending primitives that block, waiting on a reply. However, shared memory differs from these significantly, offering different benefits. As a result, most operating systems offer some form of shared memory, and choose one of message-passing or thread-tunnelling. For example, while all offer shared memory, Mach [88], QNX [47] and L4 [45] use message-passing, whereas Spring [87], Alpha [56] and Pebble [38] use thread-tunnelling. The choice between thread tunnelling and message passing is not arbitrary. There are a number of significant differences:

- Thread-tunnelling offers a lower latency than message-passing, although message-passing can improve performance where asynchronous communications are predominant (such as massively parallel systems that have a similar number of physical processors to threads).
- Message-passing is better suited to networked communications, due to physical networks' inherent message-passing nature.
- Thread-tunnelling is well suited to Remote Procedure Call [82] — the most widely used mechanism to aid distributed programming. As such, thread-tunnelling is widely employed in high-performance intra-machine RPC mechanisms, such as Lightweight RPC [9].
- Thread-tunnelling reduces the number of threads required in the system, particularly useful with fine-grained protection where most threads in a message-passing-based paradigm would be blocked, waiting on replies.
- Thread-tunnelling means that a server working on behalf of a client uses that client's thread. Therefore, requests are serviced with the scheduling and other properties of the client, avoiding QoS crosstalk [69] and reducing (although not eliminating) priority inversion. QoS

crosstalk and priority inversion happen when one thread is performing work on behalf of another — the client effectively inherits the server's priority.

Other benefits of thread-tunnelling, along with a detailed argument in its favour are presented in [35].

3.6 Security

Protection divides a computer system's resources amongst several applications. Protected domains often need to interact, one domain making requests of another. Security is the process of controlling which domains may interact, and in what ways.

There are two main aspects to security:

Authentication Securely identifying clients of services. For example the password check when a user logs in, or requiring a client provide an encrypted signature.

Authorisation Defining and implementing the policy: what clients can request what services. For example, defining that one file is world readable, while the other can be read only by its owner. There are two main mechanisms used to implement authorisation:

Capabilities A server allows any client that presents a valid capability [70] to perform certain operation(s). Different capabilities grant permission for clients to use different services. Capabilities are analogous to holding a key to a room: anyone who possesses that key may enter the room. In other words, a client's possession of some capability is both *necessary* and *sufficient* to perform the task specified by that capability. A UNIX file descriptor is an example of a capability.

Access Control Lists ACLs [98] keep a list of what clients may perform what operations. If capabilities are analogous to a key to a room, ACLs are analogous to a security guard at the door of a building who will only let people whose names are on the 'guest-list' enter.

As with the analogies given above, capabilities can be more efficient than ACLs, particularly if there are many clients of a service, but it is harder to revoke capabilities than it is permissions from an ACL.

Security cannot be implemented effectively without a protected operating system since programs that are not protected are free to bypass security. By the same token, security defines

how and where protection will be enforced. In effect, security is the *policy*, and protection the *mechanism* by which security is enforced.

Whereas security is dependent on an effective protection mechanism, the reverse is not true. While in practise some security policy is required in order for protection to be useful, it is possible to implement such security outside of the kernel. For example, Amoeba performs authentication and authorisation at user level using encryption techniques on capabilities [79].

3.7 The Trusted Computing Base

An operating system's kernel is implicitly trusted by applications not only because the kernel executes with full privileges (that is, while the processor is in kernel mode), but also because it defines and implements protection (and often security). However, many operating systems trust some code that does not run in kernel mode. For example, the UNIX system has many trusted programs, including `login`, `passwd`, `swapper` and other *daemons*. These trusted programs and the kernel form the *trusted computing base* (TCB). The TCB is the software that implements security and protection — all software that executes outside the TCB need not be trusted, although it must trust the TCB.

3.8 Summary

This chapter has introduced the mechanisms that all commodity operating systems use to employ protection. Essentially, a combination of software and hardware are used to allow several programs to run on the same computer concurrently, where each program has the illusion that it is the only program running. In protected operating systems the illusion is such that no program can be affected by another — that is, the damage that erroneous or malicious applications can cause is strictly limited to themselves.

Protection involves preventing programs from stealing others' resources, most importantly memory and CPU time. Memory protection is usually achieved by memory-management hardware (usually paging, segmentation or both), and CPU usage is restricted by using a periodic interrupt. A layer of software known as 'the kernel' manages the memory protection and interrupt hardware on behalf of applications. Applications are prevented from manipulating the protection hardware by being executed while the CPU is in a special mode of reduced privilege, known as 'user mode'.

Security is related to (but different from) protection. Essentially, security is the policy dictating

what applications may access what resources, and protection is the mechanism used to enforce this policy. Many systems use a number of user programs in conjunction with the kernel to form the *trusted computing base*. The TCB mandates the security policy that is applied to the rest of the system (that is, applications). This way the software that manages security can be cleanly separated from that which manages protection.

Chapter 4

Related Work

4.1 Introduction

Chapter 3 introduced the mechanisms used by most operating systems in order to effect protection. This chapter presents research that investigates alternative protection mechanisms.

Major OS protection paradigms are identified, both from the research community and industry. This includes the evolution of OS protection models from monolithic kernels, through early μ-kernels and software-based solutions, to the current 'state of the art' micro- and nano-kernel systems. New protection paradigms including *Object-Oriented Operating Systems* are also introduced. Note though that this chapter is *not* intended to be an exhaustive summary of all operating systems.

4.2 The Changing Role of the Kernel

Over the years the responsibilities of the kernels of subsequent generation operating systems have changed, for the most-part steadily decreasing.

4.2.1 Monolithic systems

Operating systems first existed in order to help operators manage early computer systems. OSs subsequently evolved to provide services to application programmers, and to provide the protected multiprogramming environment introduced in Chapter 3. The 'kernels' in these early systems had relatively wide responsibilities. Operating systems such as UNIX [89] included not just the

functionality that is necessarily trusted, but also abstracted hardware — that is, the hardware was presented in a form closer to that which is useful to programmers. For example, rather than presenting a hard-disk, systems such as UNIX present a file system (as shown in Figure 4.2.1).

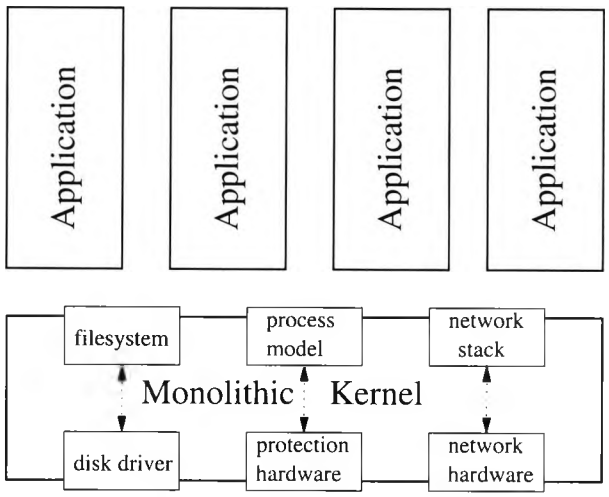


Figure 4.1: A UNIX-like monolithic kernel

4.2.2 μ-kernel Systems

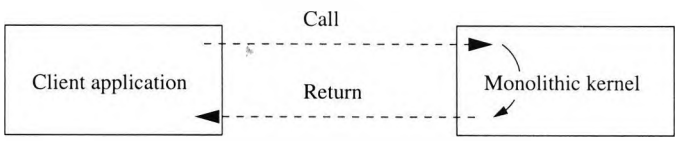
Since errors in the kernel typically lead to complete system failure, and the larger and more complicated the kernel the more chance it has of containing errors, the large kernels of monolithic systems tend to decrease system stability. Also, the static nature of the kernel means that monolithic systems tend to be less flexible — the kernel is complicated and therefore difficult to change. From the late 1980s, systems such as Amoeba [79] offered increased stability and flexibility through minimising the size of the kernel. In these systems, relatively high-level abstractions (such as file systems and networking) are not implemented by the kernel. Instead, μ-kernels are responsible for only those abstractions that provide protection directly. Thus a μ-kernel's responsibilities are limited to:

- *Resource-sharing*: multiplexing resources fairly among several competing programs.
- *Processes*: providing low-level memory management.
- *Inter-process communication (IPC)*: primitives to enable processes to communicate with one another when necessary, but in a protected fashion.

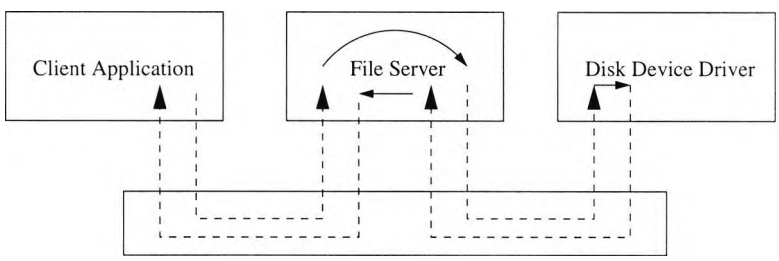
- *Device-management*: minimal structure to allow user-level programs to drive hardware devices in a protected fashion.

Higher-level services previously provided by the monolithic kernel are provided by 'server' applications. For example, file systems are not part of the kernel, but a user program acts as a 'file server'. Client applications connect to this file server via IPC, and make file system requests over such connections.

Early μ-kernel implementations had disappointing performance because of the large context-switch overhead. A client-application request on a server in a monolithic system requires two context switches — one as control passes to the kernel (thus switching context from the application to the kernel), and (once the request is serviced) another as the kernel returns control to the application. For the same operation, a μ-kernel requires at least 4 context switches: one to the kernel requesting a message be sent to the server; one to the server to service the request; one for control to return from the server to the kernel; and finally one for control to return back to the client. In fact, because the server itself is typically split into several applications, this can be multiplied many times. For example, assuming that device drivers are placed in their own address space (the ideal case for a μ-kernel) the file server might need to call a disk device-driver, increasing the context-switch count to 8 (see Figure 4.2.2). Worse still, it is likely the file server will need to call the device-driver several times per operation.



Context switching in a file operation on a monolithic system



Context switching in the equivalent operation on a μ–Kernel

Figure 4.2: Context switching in monolithic and μ-kernels

The most common technique employed to request services on a server-application is Remote Procedure Call (RPC) [82]. Here, the IPC and associated context switches required in order to invoke some service on another process are wrapped up so that they appear to the programmer as a conventional sub-routine call. RPCs can be inter-machine (that is, over a network) or intra-machine (that is, over IPC between protection domains on the same machine). If μ-kernels are not to suffer poor performance compared to their monolithic counterparts, the overheads of an intra-machine RPC must be low. The standard benchmark for such overheads is the intra-machine, null-RPC time. This is the time taken for a client application to call a procedure on a server that takes no inputs, performs no work, and produces no outputs — thus a null-RPC time measures only the *overhead* of the call.

The null-RPC times for early operating systems were disappointing. Even on a relatively modern machine (for example, a Pentium-based PC), a single null-RPC can take 100 μseconds on Linux [74]. In order to normalise against processor clock frequency, the time is often measured in 'cycles' (that is, processor clock ticks). A null-RPC on Linux takes around 47,000 cycles. Such RPC times would render μ-kernels unusable, so it is important that a μ-kernel's design allows RPC overhead to be minimised.

The Mach μ-kernel [88] was designed as a base for operating systems research. Commercial operating systems have also been built on top of the Mach μ-kernel, including NeXTSTEP [110], Apple's Rhapsody [71] and more recently MacOS X [2]. The Free Software Foundation [37] are developing a new free-ware μ-kernel based operating system, GNU Hurd [51], also based on Mach. The designers of Mach managed to get null-RPC times down to a little over 3,000 cycles [29]. While this was a considerable improvement per RPC over monolithic systems, Mach's performance was relatively poor due to the large increase in the number of RPCs for a given operation.

Attempts were made to allow server-applications to be down-loaded into the kernel in order to improve performance [21]. While this technique (known as *collocation*) was successful in improving performance, it effectively turned Mach into a monolithic system! Indeed, Microsoft's Windows NT [24] also started life as a μ-kernel. However, the transition from version 3.51 to version 4 saw the graphics library incorporated into the kernel for performance reasons (even on UNIX systems, graphics code typically exists outside the kernel). This collocation of user-level software into μ-kernels has been used as an argument that μ-kernels are unworkable. However, this conclusion is hasty at best, and only serves to show that first-generation μ-kernels performed poorly.

4.2.3 Software-Based Protection

While a combination of hardware and software is the most common technique for implementing protection, it is by no means the only one: other techniques employ purely software methods, and some even use mathematical proofs. Note that whether protection is realised through a combination of hardware and software, or software alone, a kernel is still required. Note also that it is not practical to realise protection entirely in hardware. This is partly because such hardware would be very complex, and partly because the result is likely to be inflexible. Previous attempts to realise most (or all) of protection in hardware (such as the Intel 432 [85]) demonstrated poor performance and high economic cost [20]. Of course, that previous attempts to realise protection solely in hardware have yielded disappointing results does not necessarily mean that all attempts are doomed. However, we do not know presently how to make such hardware fast, and it seems reasonable to assume that complete hardware-implemented protection will necessarily limit OS flexibility.

The poor performance of μ-kernels and the subsequent re-location of server code into the kernels caused many to loose faith in the μ-kernel philosophy. Instead, methods were explored that implemented memory protection solely in software. There have been four notable approaches: Type-Safe Languages; Software Fault Isolation; Anonymity and Proof-Carrying Code.

Type-Safe Languages

Unsafe programming languages enable arbitrary manipulation of the hardware, including access to arbitrary memory locations via unchecked pointers and unbounded arrays — hence the need for memory-protection hardware. Type-safe languages on the other hand, can be used to restrict memory access so that it is not possible for programs to access data other than their own. Operating systems are traditionally implemented in languages that are not type-safe (such as C [63]).

SPIN [8] is an operating system written using the type-safe language Modula-3. Modular-3 was intended to be a systems programming language [83], so is ideal for this purpose. In order to be suitable for systems programming it is possible to make unsafe memory accesses using Modula-3, but such modules must be explicitly declared `unsafe`. SPIN's use of a type-safe language enabled arbitrary programs to be placed in a common protection-domain as long as they were presented to the kernel as "safe" Modula-3 source code. A trusted compiler [49] is used to convert the Modula-3 source to machine code, enabling such code to be executed with confidence.

More recently, JavaOS [5] has been developed which uses the same technique as SPIN, but uses a Java [54] interpreter rather than a Modula-3 compiler to ensure protection. The use of 'Just In Time' compilation [23] can overcome the poor performance inherent in interpreted languages, and results in a system analogous to SPIN, with Java rather than Modula-3 as the trusted language (although unlike Modular-3, Java was not designed with systems programming in mind).

Language-based protection has the obvious disadvantage that all code is required to be written in a particular language. Not only is this restrictive, but it means that the system is dependent on the language's popularity amongst programmers (a problem for SPIN since Modular 3's deployment was disappointing). These systems also suffer from requiring a trusted compiler or interpreter: virtually all translators have known bugs, and these can be exploited in order to compromise system security. Even when source code is compiled to some verifiable intermediate form, such as to Java byte-code, translators and verifiers are still so complicated as to expose flaws that can be exploited in attacks [26, 48, 72, 73].

Software Fault Isolation

Software Fault Isolation (SFI) [108] uses software to enforce memory checks usually carried out by memory-management hardware. When code is loaded, the operating system surrounds all indirect memory references with a *guard*. This guard is a few machine instructions used to check the bounds of the access.

The theory is that, while SFI slows down indirect memory accesses, the time saved on context switching will more than compensate. Whether or not this assertion holds obviously depends on the ratio of indirect memory accesses to context switches. For most cases SFI has been reported to perform at least as poorly as hardware-protected systems [45]. This is partly because SFI associates a conditional branch with each memory access. Conditional branches adversely affect the performance of modern micro-processors[1].

The VINO operating system [93] allows arbitrary code to be downloaded safely into the kernel. VINO refers to code downloaded into the kernel as kernel *grafts*. Grafts are prevented from accessing memory they should not through SFI. VINO also implements a transaction system: side-effects of misbehaving grafts are 'cured' by rolling back transactions of such grafts. VINO has been successful in many regards, but its complicated transaction mechanism led to relatively poor performance.

[1] Conditional branches can cause pipeline stalls. The abundance of conditional branches is also likely to cause contention at the processor's buffers used for branch prediction and so adversely effect even those programs that make few indirect memory references.

Anonymous RPC

Anonymous RPC [112] avoids context switching (and thus the expensive hardware memory-management) by using a probabilistic approach. The technique assumes a large address space (at least 64 bit) and the availability of memory-management hardware that can mark memory as 'execute-only' (that is, the memory may be executed as code, but any attempt to read or write it will fail). This means that a client can be permitted to call a server directly, but is unable to establish the address it is calling. Assuming the machine has 4GB of memory in use, the 64-bit address space means that the chances of *guessing* (or accidentally stumbling across) an address in use by another program are $\frac{2^{64}}{2^{32}}$, which is 2^{32}, or around one in 4 billion. However, some programs rely on asking the operating system to alert them when accessing bogus addresses so that they may recover themselves (for example, when implementing copy-on-write semantics [10]). A malicious program could arrange such notification and, by repeatedly writing to random locations, expect to corrupt another program's memory in a few seconds. To render this technique unusable, the operating system can insert a delay between a program's invalid memory reference and its notification. A delay of one second would mean that a program repeatedly writing to random addresses could not expect to touch another program's memory until it had made 2^{32} writes, which would take 2^{32} seconds (over a century).

This technique suffers a number of drawbacks. Firstly, it assumes a large ratio of address space to memory used. This is an inefficient use of silicon[2], and therefore often not suitable for many systems. Furthermore, as the ratio of address-space width to memory in use continues to fall, so the odds of stumbling across another program's memory shorten. Programs that legitimately rely on being able to access invalid memory, and then be informed by the OS, suffer unacceptable performance degradation. While the above mentioned 100 year mean-time between failure is probably acceptable for desktop systems, it may not be for embedded systems. Lastly, an attack could be envisaged where a number of programs repeatedly access random memory locations concurrently, dramatically reducing the security of this system.

Proof-Carrying Code

Proof-Carrying Code [80] applies mathematical techniques and formal methods to produce a proof so that a program can be trusted in advance of its use. Unlike the type-safe languages approach, this technique can be applied to raw machine code. The proof is attached to the code, and checked

[2]Large protection domains not only require wider buses and ALUs on the processor, but wider pointers, thereby wasting memory.

when the code is loaded. The proof is constructed such that the operating system can detect any attempt to tamper with either the proof or the code, unless the code still matches the proof! Proof-carrying code is based on concepts in logic, semantics and type theory and so the details are beyond the scope of this thesis.

By the inventors' own admission in [81], this technology needs to be developed further before it is suitable for use in real OSs. Proof-Carrying Code is also somewhat restrictive, and is not applicable to all types of programs (for example, programs that use unchecked pointer arithmetic). However, Proof-Carrying Code may be a significant technique in the future, especially when combined with protection based on type-safe languages.

4.2.4 The State of the Art: Back Toward Hardware-Based Protection

In recent years a number of techniques have been used to give μ-kernel systems vastly improved performance. The latest μ-kernels appear to offer the best of all worlds: their architecture and programming are familiar to programmers, the techniques are well-understood (and so secure), and their performance is comparable to other approaches such as software-based protection or monolithic kernels. As a result, traditional hardware-based protection is once again popular.

L4: The re-emergence of the μ-Kernel

L4 is an example of a second-generation μ-kernel that offers excellent IPC performance compared with previous systems. A single IPC message can be delivered from one protection domain to another in as little as 121 cycles on the Pentium, [32], and even faster on the DEC Alpha [92]. L4 achieves this high performance through designing the entire system with the target of efficient IPC. For example, the portability at the source-code level of most first-generation μ-kernels (including Mach) is dropped, since this adversely affects performance through preventing proper exploitation of specialised hardware.

Whilst impressive in their own right, these performance figures have wider reaching implications. They prove premature the perceived wisdom that a μ-kernel (and more specifically, fine-grained protection) necessarily leads to poor performance. That is, L4 has shown that finer-grained decomposition of protection is still a worthy goal.

L4's performance still falls slightly short of the goals set out in Section 2.3.2. Also, the figures reported (the single IPC message delivery times) are also not necessarily the most interesting ones. Most programs communicate using remote procedure call, and so the complete null-RPC

time is more interesting than raw IPC-message delivery times. A complete null-RPC on L4 on the Pentium has been measured at approximately 280 cycles. (Still an impressive figure when compared to its contemporaries, such as Mach's figure of 3,000 cycles!)

Exokernels

Recently, MIT has developed the notion of *exokernels* [30]. Exokernels have just one responsibility: to multiplex hardware resources amongst applications in a protected manner. An exokernel does this by presenting hardware with barely any abstraction. To affect protection, an exokernel tracks resource ownership and employs some 'abort protocol', used to terminate misbehaving programs.

The motivation for exokernels was that no matter what abstractions an operating system provides, they are likely to be inappropriate for many applications. The solution: eliminate all OS abstractions. For example, there are no notions of processes or threads; just CPU time-slices and protection domains. Libraries provide applications with the abstractions they expect from operating systems such as threads, processes and inter-process communication. Such libraries are referred to as 'Library Operating Systems'.

One might expect that reducing the kernel to an absolute minimum would compound the performance problems encountered by μ-kernels. However, exokernels have demonstrated *improved* performance over even monolithic systems. Experiments with exokernels have demonstrated significant improvements in application performance. For example, the *Cheetah* web-server performs up to 8 times better than web-servers on monolithic kernels [58].

Exokernels have proved a popular approach among researchers. There have been several exokernel-based implementations, including the *Aegis* and *Xok* systems [31] (developed as prototype exokernels that run on MIPS-Based DEC5000 machines and IA32-Based PCs respectively). *ExOS* is a prototype library-operating system that sits on top of both Aegis and Xok to provide the applications with UNIX-like services. The *Charm* Operating System [27] is a separate development effort that uses the exokernel approach to provide a basis for a persistent operating system.

Cache Kernels

When developing the successor to the V μ-kernel[17], V++ [18], Cheriton et. al. also striven to minimise the detrimental effects of inappropriate OS abstractions. Unlike the exokernel approach, the V++ kernel preserved some abstractions, such as threads, but allowed applications

(or library operating systems) fine-grained control over the multiplexing of these abstractions onto hardware resources. This was accomplished by calling on the application owning the resource to be responsible for storing that resource away while it is not currently executing on the hardware. For example, when a thread is preempted from the CPU, the owning application is called upon to save the thread's interrupted context. Some time later, the application will be called upon to reload a thread, when it may chose that, or another thread. The same technique is applied to other multiplexed abstractions such as protection domains and physical memory. Effectively, multiplexed resources are managed by their owning applications, and resources such as threads and protection domains are *cached* inside the kernel when they are active.

The cache-kernel approach, though successful, has not proven as popular amongst the research community as the exokernel one. This is mainly due to exokernels' simplicity and greater flexibility.

4.3 Object-Oriented Operating Systems

Over the previous decade, object-orientation has become the software development methodology of choice[3]. There have been several projects to produce object-oriented operating systems (or OOOSs). This section identifies the different categories of each OOOS and explores at least one implementation of each.

4.3.1 Object-Supporting Kernels

Much of the earlier OOOS work focused on traditional OS kernels, with primitives aimed at supporting object-based software. Clouds [25] was such an OS: a monolithic kernel providing object-based services. Later version of Clouds employed a μ-kernel (called *Ra* [7]) to provide OO services. Objects in Clouds were heavy-weight, consuming a comparable amount of resource to UNIX processes. As a result, object-based systems built on top of Clouds needed to be relatively course grained if poor performance was to be avoided.

4.3.2 Object-Oriented Kernels

Object-oriented kernels, such as CHOICES (Class Hierarchical Open Interface for Custom Embedded Systems[4]) [13] are the opposite of object-supporting kernels: they are kernels providing

[3] A discussion of the details of object-oriented programming (OOP) is beyond the scope of this thesis (see [12] for a detailed description of OOP).

[4] Possibly one of the most contrived acronyms in computer science!

traditional OS services, but implemented using an object-oriented approach (usually using C++). The motivation for Choices was 'to see if it could be done' — that is, are OO software development techniques applicable to kernel design? Given the motivation, the results were encouraging: Choices showed that OO software engineering techniques were indeed applicable to kernel development.

Since CHOICES, there have been several commercial implementations of object-oriented kernels, including Symbian's EPOC-32 [102]. This is an embedded system, first used in 'Palm Top' computers, but now adopted by several mobile-device manufacturers such as mobile-phone companies.

4.3.3 Object-Based Systems

After the success of object-supporting kernels, and object-oriented kernels, OSs were developed to combine the two: that is, object-oriented kernels that provide support for object-oriented applications.

Spring

Sun Microsystem's *Spring* [87] is an example of an object-based operating system. All objects were specified in an Interface Definition Language (IDL) and OO techniques such as inheritance were employed throughout the system [42]. Spring boasted an identical object-model inside its μ-kernel and out. However, the system call required to traverse privilege modes meant that the way an object was accessed depended on the processor mode in which it resided[5].

Despite impressive IPC times (600 cycles for a cross-domain control transfer), Spring still did not mandate protection between all objects in order to maintain performance and allow objects to be relatively fine-grained (for example, one object per file). That is, protection was implemented at a level above objects: several objects would be grouped together into a *domain*. Even so, performance was disappointing due to the still large number of RPCs — a similar problem as with most μ-kernels of that era. As a result, Spring never made it out of the lab.

KeyKOS

KeyKOS [44] is another example of a research object-based operating system. It has an elegant design whereby objects are accessed via *keys* (KeyKOS's term for a capability [70]). The invocation

[5]The distinction between user and kernel mode can be hidden by proxies.

of a key is effectively a method call on the object referenced by that capability. The server is delivered a *resume key* which it may employ to return to its caller, effectively providing standard sub-routine linkage.

Because conventional access-control-list protected file systems give up the advantages of capability-based systems, KeyKOS uses a *single-level store* to implement persistence. Here, demand-paged virtual memory is used to provide a transparently persistent system. To cope with bugs in the kernel a 'snap-shot' mechanism is used to allow the system to be rolled back to a known consistent state.

EROS (Extremely Reliable Operating System) [94] is a recent re-implementation of KeyKOS. EROS introduces a new form of capability: the *weak* capability. When weak capabilities are passed to other principles their power is reduced. This entity has been used to verify formally EROS's confinement mechanism [95].

4.4 Component-Based Operating Systems

This section introduces the most recent *genre* of operating systems: component-based operating systems. Unfortunately, there is no universally agreed-upon definition of the terms 'component' or 'object', still less their differences, and so this section begins by defining some terminology. A good differentiator is that objects are fundamentally a programmers' tool while components are more concrete entities. That is, traditional objects exist in the program's source code only, and are pertinent mainly to type-theory. For example, once a C++ program is compiled, the boundaries between objects disappear; indeed, it is not possible to state with 100% confidence whether a binary was produced using C++ or C as its source (or even assembly). On the other hand, the boundaries between components are concrete and are present in a running system — it should be trivial to produce a tool to allow the user to examine what components exist at a any time. In this regard, a traditional file is closer to a component than is an object. In fact, a process is a better analogy still since a process includes behaviour as well as state.

By the above definition of a component, most object supporting and object-based operating systems can be considered component-based in all but name. In fact, component-based systems are even older than this: in some senses UNIX is component-based (albeit at a very course granularity). That is, a UNIX system is built from several components, namely: the kernel; utility programs such as the shell, `cp` and `ls`; device drivers; the `login` process; X-Windows; virtual file systems, and so on. Such decomposition of operating systems is so universally accepted as good practise

that UNIX is not usually considered a component-based OS!

The challenge facing component-based OS researchers is to decompose systems at a finer granularity, and with more rigour so as to compound the well-accepted benefits of UNIX-style, coarse-grained decomposition. That is, to decompose the kernel into components such as schedulers, memory managers and pagers. A component-based operating system is defined here as a system with a component model that exhibits the following properties:

Explicit. Components are clearly separated, possibly even with each component residing in its own protection domain. All component interaction is via well-defined interfaces, ideally specified in some semi-formal manner (such as with an IDL file).

Orthogonal and Consistent. The nature of a component should be independent of what tasks the component performs, and all components should look the same throughout the system. For example, UNIX cannot be considered truly component based because the nature of a file is determined by its function (to store data). Moreover, there are many different kinds of components; for example: files, processes, the kernel, utilities, shared libraries and device drivers.

Pervasive. The entire system should be built from components, from system-level functionalities such as interrupt dispatching and memory management, to higher level services such as user interfaces and spell checkers.

Lightweight. The component model should be lightweight to allow for fine-grained decomposition of services. This will mean that services can be decomposed according to design rather than performance considerations.

Such decomposition results in operating systems that are highly configurable, robust, employ modern software-engineering techniques, and (due to increased flexibility [58]) perform well.

SawMill

One of the most notable research efforts into component-based operating systems is SawMill [55], developed at IBM's T.J. Watson laboratory. SawMill is based on the L4 μ-kernel to offer fine-grained protection while maintaining performance. While components are still bundled together into protection domains (as with Spring), they are free to migrate between protection domains. That is, SawMill aims to provide decomposition at a fine granularity by developing a new security

model rather than a new protection model. However, this strategy introduces complicated security issues, and much of the SawMill development effort is spent investigating security policies [104].

Pebble

Pebble [38] is another component-based OS development effort currently under way at Bell Labs. Here, each component exists in its own protection domain, avoiding the security issues raised by SawMill. The kernel itself, although small, is not decomposed (that is, the kernel is an amorphous mass of code which performs several quite separate tasks such as inter-component communication and interrupt handling).

Protection is implemented traditionally (using paging) and so Pebble is of little relevance to the work presented here, other than being component based.

4.5 QoS and Protection Guarantees

This section introduces protection paradigms used to provide improved *guarantees*, especially with regards to protection of the CPU. While all protection paradigms provide some guarantee that programs will not be able to steal *all* of another's CPU time, exactly how much CPU each program will receive is usually not specified. That is, the OS can guarantee that a program will get *some* CPU time, but exactly how much depends on what other programs are running and what they are doing. Compared to resources such as memory, CPU division and protection is somewhat arbitrary in conventional, desktop OSs.

Several OSs are addressing this problem, particularly with the advent of desktop machines capable of handling continuous, streaming media (such as video and audio). A summary of the most interesting techniques used to divide CPU time more accurately are given in the remainder of this section.

4.5.1 Conventional Real-Time Operating Systems

Real-time systems are those that must not only compute the right results in order to be correct, but the result must be produced before some deadline. This is often referred to as *predictability*. Conventional, desktop OSs' scheduling (such as UNIX's) is not predictable: instead these systems aim to maximise *throughput* — that is, the OS is built to get as much as possible done every second. Maximising throughput means that the OS attempts to achieve the best overall performance, even

if this is at the expense of having performance occasionally dipping to a relatively low level. On the other hand, predictability means that the system must achieve some minimum performance, always.

Conventional real-time systems achieve these guarantees by using a simple scheduling policy. Each thread has some priority, such that the runnable thread with the highest priority is always scheduled. If each thread has a different priority, mathematical techniques can be used to prove that a system will always meet its dead-lines. This technique is called Rate Monotonic Analysis (RMA) [106]. However, RMA applies only to systems that are relatively static — usually embedded systems where some fixed number of threads perform a well-known set of tasks. RMA is not applicable to more complicated systems (such as desktop systems), where the designer cannot know in advance what programs will be running.

4.5.2 QoS Scheduling

QoS (Quality of Service) allows dynamic systems to provide the sort of real-time guarantees usually associated with static, embedded systems. The main obstacle to dynamic systems meeting real-time guarantees is *"QoS crosstalk"*. This refers to the situation where tasks are performed by a collaboration of protection domains, and some domains perform part of more than one task. For example, imagine two programs, A and B, are executing on a system that implements demand-paged virtual memory. Program A is of low priority, frequently accessing pages that are swapped out, while program B is of high priority, and accesses only pages that are resident in main memory. The operating system will have to service many page faults (and swaps) on behalf of A, reducing the amount of CPU received by program B — despite protection, program A has stolen some of program B's resource. It is for this reason that many real-time systems do not support demand-paged virtual memory.

As another example of QoS crosstalk, imagine a message-based system where two clients, A and B, both use some server, S. Even if client A is of a lower priority than B, if A makes heavy use of S, it will take resource away from B.

The remainder of this section introduces research OSs that aim to overcome QoS crosstalk.

Nemesis

Nemesis [69] avoids QoS crosstalk by implementing a *vertically integrated* system. This means that communication between protection domains is limited by having each domain contain an almost

complete OS: in effect each application has its own library operating system.

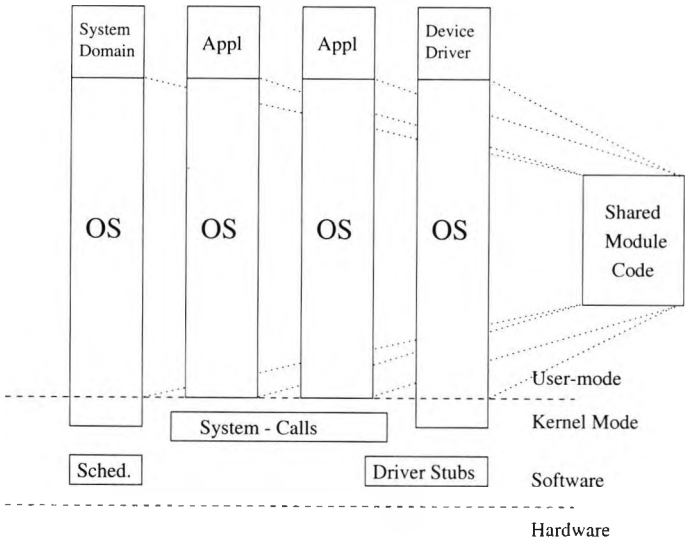


Figure 4.3: Vertical system integration in Nemesis

The theory is that the notion of multiplexing different programs onto one computer is taken to its logical conclusion in order to eliminate QoS crosstalk. For example, imagine two physically separate and totally isolated computers, each running a single application. It is obvious that one application will not affect the performance of the other. Nemesis aims to model this total separation on a single machine. This is achieved by duplicating the entire system, OS included, for each application.

Nemesis's vertical integration is shown in Figure 4.3. Note that each separate operating system is in reality shared code — the vertically integrated system has little or no higher memory requirements than conventional systems.

When several programs are multiplexed onto one computer some degree of QoS crosstalk is inevitable; for one thing, asynchronous interrupts and hard QoS guarantees are mutually exclusive. However, Nemesis' vertical integration lowers the level at which multiplexing occurs: the smaller the part of the system that is shared, the smaller the potential for QoS crosstalk. Hard disk access [6] and even demand-paged virtual memory [43] have been implemented on Nemesis with no significant QoS crosstalk.

Scout

Like Nemesis, the Scout [78] operating system aims to reduce or eliminate QoS crosstalk. However, Scout does not rely on vertical integration; instead Scout introduces a new abstraction: *paths*.

Scout is essentially a component-based operating system, although Scout components are referred to as *modules*. Modules may reside in their own protection domain, or share protection domains in a configurable fashion. The designers of Scout made the observation that to avoid QoS crosstalk the system should not use modules or protection domains as the schedulable entity: instead one should schedule *tasks*. A task can be viewed as a set of interconnected modules, known as a *module-graph*. An example module-graph is shown in Figure 4.4.

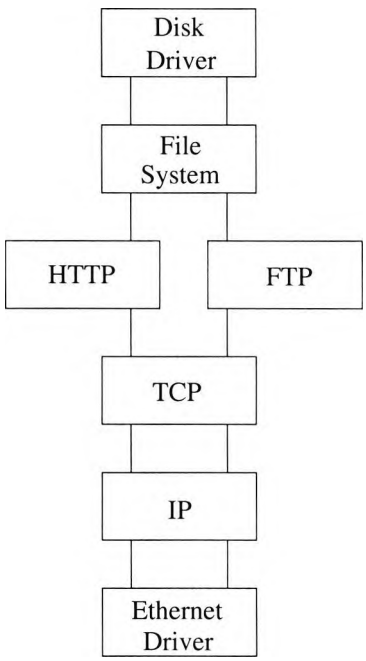


Figure 4.4: Module graphs and paths in Scout

The scheduled entity in Scout is a complete *path* (*path* in this context means a series of adjacent edges). The example in Figure 4.4 shows module graphs for a web-server and an FTP server. Different modules implement different protocol layers, the file system and disk device driver. Note that the FTP and WWW paths are identical with the exception of the top-most protocol layer. If *modules* were the schedulable entity then it is clear how QoS crosstalk would occur between FTP and WWW servers. Instead, because *paths* are the schedulable entity, the WWW and FTP servers can be assigned different priorities without crosstalk. Also, demultiplexing incoming network packets early in a path can be used to limit "denial of service attacks" [96]. That is, the affects of attempts to bring down a host by bombarding it with network packets can be reduced dramatically (although the problem is still not completely 'solved').

AlphaOS

AlphaOS is a more conventional real-time operating system [56], developed at Carnegie-Mellon University (not be confused with DEC's Alpha microprocessor). AlphaOS avoids QoS crosstalk by using the simplest solution: thread tunnelling (see Section 3.5). Scout and Nemesis designers rejected thread tunnelling because it complicates user-level scheduling policies.

QoS and Component-Based Systems

QoS crosstalk describes the situation where one protection domain steals resources from another indirectly. One way to eliminate QoS crosstalk is to instantiate each process on a physically separate and totally isolated machine. This is undesirable for many reasons, and so the Nemesis system provides this separation in software through vertical integration. This reduces QoS crosstalk because essentially, the less of a system that is shared, the less the potential for QoS crosstalk. The inverse must therefore be true: the more of a system that is shared, the greater the potential for QoS crosstalk. Since in a component-based system more work is shared than in conventional systems, all other things being equal, a component-based system is more susceptible to QoS crosstalk.

Unfortunately, the complete separation of services implemented in Nemesis is directly contrary to decomposition, code re-use and inter-operation that component-based systems strive for. That is, more is shared in component-based systems. Thus the policies implemented by Scout and AlphaOS are more appropriate for the elimination of QoS crosstalk in component-based systems than those of Nemesis.

4.6 Summary

Over the years, OS research has led to (on the whole) progressively smaller kernels. The early and mid-nineties saw a brief deviation from hardware-based protection to language-based protection because of perceived performance barriers. However, such perceptions have been shown to be overly pessimistic, with recent hardware-based protection systems out-performing language-based ones. Also, doubts have been cast on the feasibility of language-based protection meeting security requirements. Consequently, state-of-the-art operating systems base protection on hardware once again.

Simultaneously, there have been several efforts to base OSs on objects, with the object-

orientation becoming stronger, and a more integral part of the OS — from traditional kernels developed in C++, to objects (or *components*) becoming the system's principle abstraction.

Different object-models inside and outside the kernel complicate the programming model, and the system as a whole, by elevating what is essentially an implementation detail to the interface (that is, in which processor mode a component executes effects how a component is called).

If possible, the elimination of separate processor modes while maintaining protection promises to provide significant advances in all of the areas highlighted in section 2.3. However, to be truly useful this elimination of separate processor modes must not place constraints on software engineering, such as dictating the use of a particular high-level language.

Chapter 5

SISR: A New Protection Model

5.1 Introduction

This chapter introduces a new OS protection mechanism, SISR (Software-based Instruction Set Reduction). After examining why the protection models presented in Chapter 4 are unsuitable for realising the goals outlined in Chapter 2, the new protection model is described (the crux of this thesis).

Essentially, this model restricts the use of privileged instructions not through separate processor modes, but by scanning untrusted code prior to its execution. That is, all code executes while the processor is in kernel-mode, but the system rejects any user-level components found to contain privileged instructions. Code-scanning works on the premise that if a component's code does not contain any privileged instructions then that component cannot execute any privilege instructions (providing of course that all code is in read-only memory).

Although code-scanning is effective at preventing the execution of privilege instructions, it cannot prevent illegal memory accesses (memory protection), nor cannot it solve the halting problem (CPU protection). SISR implements memory and CPU protection through segmentation and interrupts respectively.

Code-scanning is also used to ensure that segment registers are not loaded with arbitrary values. That is, segment-register loads are considered a privileged operation; the code-scanner rejects untrusted code sections that contain instructions that cause the loading of a segment register. Segmentation is used to provide memory protection since contexts are switch using explicit instructions that load segment registers meaning that the code scanner can trivially identify

protection-context traversal.

All inter-component method calls are indirected via a special component: the Object Request Broker (ORB). The ORB is responsible for allowing components to call one another in a protected fashion; it loads segment registers on behalf of components that wish to call another.

Executing all code in kernel-mode has several advantages. Most notably the temporal performance of a context switch is improved dramatically because a few segment-register loads are required, rather than trapping to kernel mode, altering page tables (potentially invalidating the TLB), and switching back to user mode, as in traditional protection models. There are several other advantages to this model including simplicity, low spatial overheads and flexibility that all help to meet the goals outlined in Chapter 2.

Note that SISR is a *protection* model, and does nothing to provide *security*. Security policies are delegated to components inside the SISR system — that is, security policy is properly decomposed from the protection mechanism.

5.2 Protection with SISR

To achieve the performance goals cited in Section 2.3.2, it is clear that traditional protection is inadequate. This section examines the reasons behind traditional protection's poor performance and inflexibility, and presents a new protection paradigm. This new paradigm promises to deliver the required performance improvements while maintaining language independence. The low overheads imposed by this new model lead to protection being practical at a much finer granularity than with traditional protection. This fine-grained protection has the potential to lead to considerable improvements in software engineering, dynamism, and configurability. Flexibility depends on implementation (see Chapter 2), but SISR provides for systems that are at least as flexible as exokernels.

5.2.1 Memory Protection

Traditional memory protection is based on paging, requiring that each protection domain must be a multiple of the page size. This means that if paging is used, each protection domain will loose on average half the page size to internal fragmentation (because the context's memory size must be rounded up to an exact multiple of the page size). This means that even in the leanest design, an average of $\frac{1}{2}nP$ bytes are lost to internal fragmentation (where n is the number of protected

domains and P the page-size in bytes). The finer grained the protection, the larger n becomes. The system outlined in Section 2.3.1 will have fine grained protection, so n will be large. Since pages are typically several kilobytes in size, paging will result in unacceptable spatial overheads, violating the space requirements specified in Section 2.3.2. Furthermore, the requirement for language independence (Section 2.3.5) renders language-based protection techniques unsuitable.

Due to these space and language requirements segmentation is used to effect memory protection. Each component's data reside in one data segment, and a component's code resides in a code segment. Each component instance has its own data segment, but components of the same type share a code segment. There is also one stack segment per thread.

As mentioned in Section 5.1, segmentation is also used to effect memory protection because it allows the code scanner trivially to identify context switches.

5.2.2 Privileged Instruction Protection

As identified in Chapter 3, protection cannot be implemented by memory protection alone, but also requires that untrusted code is prevented from executing privileged instructions. This section examines the limitations of traditional techniques, and presents a novel but simple technique to prevent inappropriate execution of privileged instructions.

Limitations of Separate Processor Modes

As described in Section 3.3.3, untrusted programs are traditionally prevented from executing privileged instructions by the use of a separate processor mode of reduced privilege. Any system with multiple privilege-levels must be able to switch between these levels. These switches are temporally expensive, typically requiring many dozens of cycles[1]. The overheads of this privilege-level switching are compounded since each RPC requires 4 such transitions (see Figure 4.2.2). The DEC Alpha [22] is an example of a very fast architecture in this regard, requiring just 8 cycles per transition: most architectures impose a far higher penalty (usually around an order of magnitude higher). Even on the Alpha, 32 cycles per RPC (for the 4 privilege-level transitions required) is a significant amount of the 150 cycle limit imposed in Section 2.3.2.

There is a less obvious reason why separate processor modes are not well-suited to component-based systems. This separation of user and kernel mode (referred to from here on in as the 'system-call barrier') works well for traditional kernel operating systems. However, in systems

[1] An intra-domain function-call will typically take less than 10 cycles. Inter-domain calls impose orders of magnitude more overhead.

that use an identical object-model in both user and kernel mode (such as Spring [87]), the system-call barrier renders the object-model of each side invisible to the other. This means that many of the benefits of object-oriented applications are denied to the kernel, and vice-versa. Modern component-based operating systems work hard to hide the distinction between user mode and kernel mode[2].

The system-call barrier imposes both architectural and performance bottlenecks on component-based systems and thus is an inappropriate protection mechanism.

Catching Privileged Instructions using Code-Scanning

With SISR, all code executes in a single processor mode (with full privileges). However, code that is not trusted is prevented from containing any privileged instructions. This is achieved by scanning code prior to its loading, and rejecting any code containing instructions that it is not sufficiently privileged to execute. This code-scanning will not impose significant overheads: only a few cycles are required per instruction. When one considers that this code will need to be loaded from a source typically orders of magnitude slower than modern micro-processors (such as a disk or network), such analysis can be seen to impose no significant penalty. Furthermore, the scanning need only be performed when a new type of component is installed on the system, not each time an instance of a component is created. For example, the file system could use a bit that is set when the file contains a component image that has been scanned, similar to the UNIX `setuid` bit [89] (assuming that the file system is trusted by the security mechanism). Note that it is similarly not necessary to scan code sections which are loaded during demand-paged virtual memory since whatever components provide demand paging are necessarily trusted.

In some ways code-scanning is similar to Java's bytecode verification [77], since binary code is scanned prior to installation to ensure that it is 'safe'. There are however, many differences. Firstly and most obviously, code scanning works with native machine code and so is language-independent (as required in Section 2.3.5). Secondly, because the byte-code verifier does not rely on memory-management hardware, many types of memory access cannot be reliably verified and so must be prevented by the source language (for example, unbounded array accesses or pointer arithmetic). Lastly, code-scanning is much simpler than bytecode verification: code-scanning requires searching for certain bit-patterns in some binary code, whereas bytecode verification requires several 'passes', one of which performs data-flow analysis of the code.

[2]Whether a component executes in kernel mode or user mode is of concern only to its implementation. It should not be necessary or possible for a component's client to determine in which processor mode it operates, since this is of no consequence to the interface.

At first glance, code scanning can appear similar to software fault isolation (SFI) [108]. On closer inspection, the two techniques can be seen to be quite different. SFI inserts instructions into object code in order to effect memory protection. Code scanning works with unmodified binaries[3], scanning them for the presence of certain instructions. SFI effects memory protection through software. SISR uses segmentation hardware to effect memory protection, and code-scanning to provide privileged-instruction protection. The two techniques are similar only in so far that they both provide protection through working with code prior to its loading.

Context Switching

If the code-scanning technique is extended to prevent applications loading segment registers with arbitrary values then holding a segment's selector in a segment register can be considered a capability to access that segment [61]. Thus context switching is made much more efficient: rather than the usually expensive manipulation of complicated page/descriptor-tables to effect a context switch, the code, data and stack segment registers are loaded with new values.

The time saved on switching processor modes (identified in Section 5.2.2) is only a small part of SISR's efficiency. Since code-scanning allows the operating system to define what instructions are privileged, SISR's main speed advantage comes from defining segment-register loads as privileged instructions — a context switch can now be effected by simply loading a few segment registers.

Code Containment

Code is prevented from calling arbitrarily into other protection domains because inter-segment branches are classed as segment-register loads and so prohibited by code-scanning. However, almost all components will need to able to communicate with others, at least via the TCB. In order to allow components to call the TCB at well-known entry-points, a few immediate inter-segment branches must be permitted by the code-scanner. For example, assuming the TCB's code segment is addressed by selector 0, the instruction `call 0:0` might be permitted by the scanner. This instruction will call the TCB's first instruction, transferring control to a well-known entry-point and swapping contexts atomically: the analogue of a system call. All other inter-segment branches (including indirect) are treated as normal segment-register loads and so prohibited.

Note that this property of segmented architectures is necessary if code-scanning is to work well. That is, unlike with page-based protection, explicit segment-register load instructions make it

[3]Some modification of binaries is required in order for code scanning to work with variable-length instruction sets. See Section 5.7.2.

trivial for the code-scanner to identify code that refers to another protection domain. Furthermore, assuming that the instruction-set architecture allows immediate inter-segment branches (such as `call 0:0`), it is trivial for the code scanner to allow a few, specific ways for a component to call into another protection domain. Since page-based memory protection does not have this property, using code scanning on a page-based architecture is much more complicated and expensive than with segmented architectures (see Section 5.7.2 for details).

5.2.3 CPU Protection

SISR provides nothing new with respect to CPU protection: interrupts are used as normal. That is, callers must arrange to respond to some 'watchdog' time-out if they do not trust their callee to return. Along with thread preemption, such time-outs can be implemented by responding to a timer interrupts. Note that this is no different from traditional protection models' solution to the halting problem.

5.2.4 Summary of Section

In summary, SISR has the advantages of (a) reducing dramatically the usually high penalties of context switching, and (b) simplifying architectures by removing the 'system-call barrier'. The technique is also more flexible than catching privilege instructions using hardware: certain code sections can be allowed restricted use of some privileged instructions but not others (for example, a device driver might be permitted use of certain I/O control instructions but not the ability to disable interrupts).

In the unlikely event that support is required for components that cannot satisfy code-scanning (for example, self-modifying code is required) those few components can be executed in a less privileged mode. Although such components would suffer the associated performance penalties, the effects on the rest of the system would be minimal.

5.3 Components

In SISR a component is a code segment with a data segment and a table of method entry-points known as its *Method-Table*. There is a code segment and method-table per implementation type, as well as an *initial data segment*. There is one data segment per instance of that implementation. This means that a component instance can be identified by its data segment selector, and a type

by its code segment selector. This model of component types and instances is similar to that found in other object-based systems such as KeyKOS [44] and Monads [61].

5.4 The Object Request Broker

As mentioned above, nearly all components will need to call upon the services of others. This requires a context switch (from the caller to the callee), which is effected in SISR by loading new values into the stack, data, and code segment registers. However, if protection is to be maintained, the segment-register loads required for context switches must be controlled. Code-scanning cannot easily distinguish between the legitimate loading of segment-registers to effect a context switch, and malicious behaviour to gain illegal access to another component.

To ensure that contexts are switched in a protected fashion, some trusted, generic, context-switching code is used. This trusted code is known as the Object Request Broker (ORB). The ORB is similar to a CORBA ORB [41], although it does not adhere to the standard (that is, the ORB fulfils a similar role to a CORBA ORB, but the details are different). This is shown graphically in Figure 5.1.

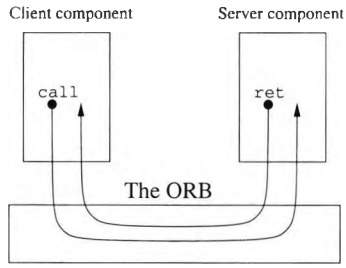


Figure 5.1: Calling via the Object Request Broker

If the ORB's code exists in a segment with a well-known selector, untrusted code can then call into it at well-known entry-points using immediate inter-segment calls. For example, assume that the context-switching code resides in the segment identified by selector `0`. Assume also that the context-switching code has the following entry-points:

`call` at offset `0` which causes the context to switch to a callee component, and control to jump into that component's code segment at an offset specified by its method table.

`return` at offset `38h` which causes a return to caller. That is, the context is switched back to the most recent component to invoke `call`, returning as if from a conventional function call.

In order to call into another context, the caller issues the instruction `call 0:0` with the target component identified by the contents of a general-purpose register. In order to return from a previous call, the callee executes `jmp 0:38h`. As stated previously, code-scanning will prevent the loading of any code section that contains instructions to load a segment register. However, in order for untrusted code to able to call upon the ORB, there are necessarily a few exceptions to this rule. The code scanner allows code that contains immediate inter-segment calls to valid ORB entry points. For the example implementation given here, code sections may contain instructions `call 0:0` and `jmp 0:38h`.

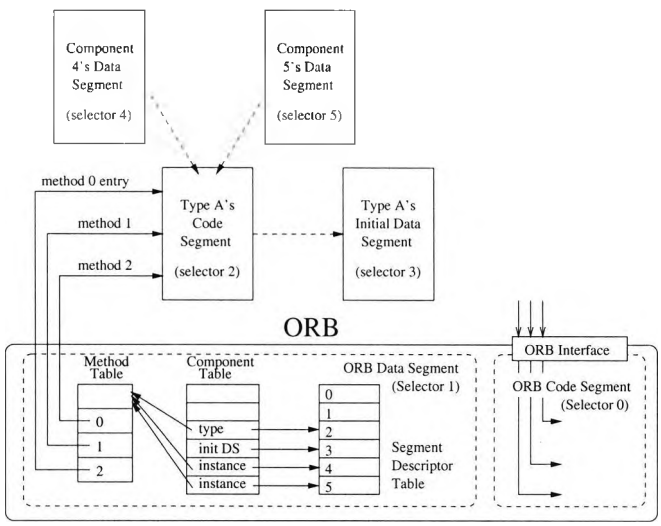


Figure 5.2: The Object Request Broker, method-tables, and segments

Figure 5.2 shows a simple example. There are two components (numbered 4 and 5), both of type A. The ORB contains A's method-table, which points at the entry-points of each of type A's 3 methods. The ORB also contains the Segment-Descriptor-Table, which describes the data segments for components 4 and 5, the code and initial-data segments of type A, and the ORB's code and data segments.

5.4.1 ORB Implementation Details

In order to explain more accurately the ORB's behaviour, an algorithm for the `call` primitive is given below:

1. push data-segment selector onto stack

2. load ORB data-segment selector into data-segment register

3. validate callee component reference and method number
4. push details of previous call onto the stack (e.g. previous stack size)
5. increment callee's call-count
6. shrink stack by manipulating segment-descriptor-tables (assuming stack grows towards zero, set `descriptor_table[ stack_segment ].limit` to the value of current stack pointer)
7. increment call-depth associated with current stack segment
8. look-up callee code segment and offset in method table
9. load callee data segment into the data-segment register
10. place callee data segment in general-purpose register (for authentication)
11. jump into callee code segment at offset indicated by method table

And to return:

1. load ORB data-segment selector into data-segment register
2. re-grow stack by manipulating segment-descriptor tables to refer to the previous stack's size
3. pop details of previous call from stack (e.g. previous stack size)
4. decrement call-depth associated with current stack segment
5. decrement callee component's call count
6. pop caller's data-segment selector into data-segment register
7. issue an inter-segment return, returning control to caller

Note that the callee must be denied access to the caller's stack-frame if protection is to be maintained. Note also that the data-segment selector of the caller is passed to the callee. This is how the ORB performs *authentication* (that is, securely identifying a client to a server). Unlike most operating systems, the ORB does not provide authorisation: it is left to servers to determine what clients are able to perform what operations (perhaps using capabilities or ACLs). Server components can either perform authentication directly, or plug into some authorisation mechanism built on top of the ORB: security policy (specifically, authentication) is component-based too (that is, the protection and security mechanisms are decomposed).

If required, (preemptive) multithreading can be implemented by components outside of the ORB. The above algorithm is implicitly thread safe since all manipulated variables are associated with the current stack segment, which is thread specific.

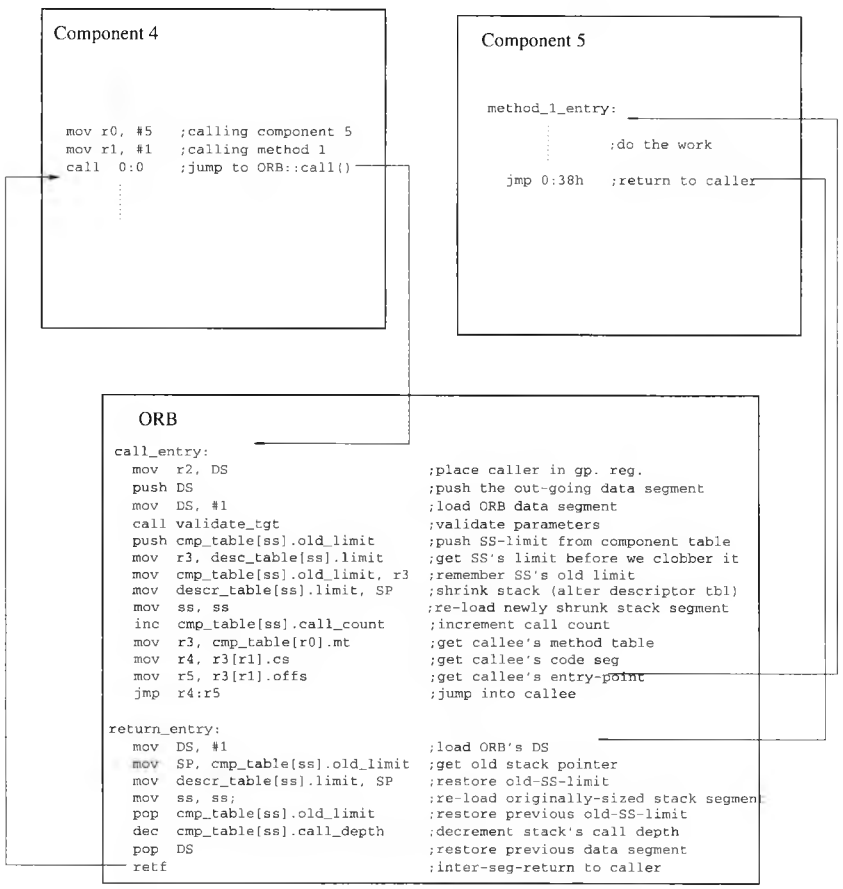


Figure 5.3: Calling a method on another component via the ORB

Figure 5.3 shows a more complete implementation of the ORB's `call` and `return` methods using pseudo-assembly language.

5.5 Redressing the TCB

As stated in Section 2.3.1, it is desirable to decompose the TCB itself. The use of separate processor modes in traditional systems restricts decomposition of the TCB (or at least makes TCB components special, since they are required to execute in a different processor mode). While the TCB needs to be involved in all implicitly trusted operations (such as interrupt/fault handling, thread preemption, paging and code-scanning), these operations should ideally be separated into

different components. Furthermore, if the architecture is to be decomposed yet simple, these components should not differ from components outside the TCB[4].

The single processor mode used with code-scanning allows components to perform system-level tasks such as interrupt-management, providing that the code-scanner knows to trust such components. Note that the code-scanner might be programmed statically to know which components to trust, might know to trust all components installed before it, or might need to be told by the system administrator which components to trust. The point is that unlike with most systems, which components form the TCB is not mandated by the protection model. In this new model, the ORB handles component-management only — that is, creation, destruction, (un)installation and inter-component method invocation/return.

Since everything executes in kernel mode, one might argue everything is downloaded into the kernel. However, the concept is quite different: there is no kernel/user mode divide at all; one could equally argue that all code is uploaded out of the kernel.

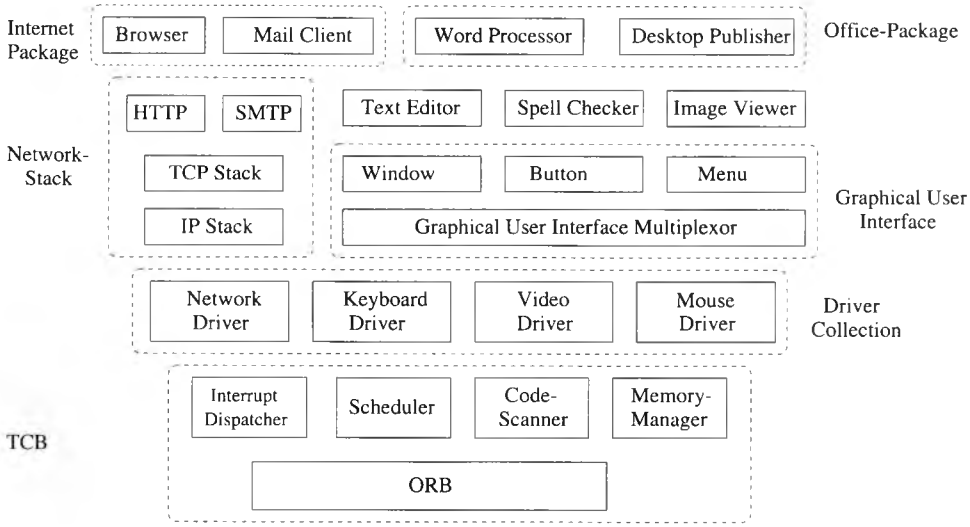


Figure 5.4: Protection is just a service provided by several cooperating components

Figure 5.4 shows an example collection of components. Different components group together to provide a service on which the components comprising the layers above rely. In this respect, the *TCB* is no different from the *network-stack* or *office-package*. The only special component is the ORB, which, since its chief role is to mediate inter-component invocation, must be called differently (otherwise calling the ORB will be recursive).

[4]Being part of the TCB (and therefore being trusted) might be argued to make a component special. However, there is no concrete difference between a set of components cooperating to provide a TCB than a set providing any other service such as a graphical user interface, or an e-mail client.

5.6 Performance

Presuming a competent implementation (such as the one suggested in Section 5.4.1), SISR should lead to dramatically improved performance on a context switch. Note though that the improved performance is not an end in itself: the improved performance allows increased decomposition, where components are protected from one another. Other systems are forced to have either very course-grained decomposition, or allow *collocation* of components into one protection context in order to maintain performance. Such collocation is nearly always done for performance rather than architectural reasons: if two components need to trust each other for architectural reasons then those components should probably be collapsed into one. The low overheads of SISR means that trust can be pessimistic — components need not trust any other components (apart from those comprising the TCB). In other words, encapsulation can be enforced by the protection mechanism while reasonable performance is maintained.

5.7 Subtleties

This section introduces the limitations of SISR, and explains how they can be overcome. There are three 'issues': self-modifying code, variable-length instruction sets, and page-based protection. These subtleties are explored in more depth in the remainder of this section.

5.7.1 Self-Modifying Code

Code scanning is incompatible with self-modifying code, since such code could break protection by writing privileged instructions to itself after code-scanning has been performed. It also prevents arbitrary data being embedded in code sections, because such data might appear to the code-scanner to be privileged instructions. However, both these techniques are rarely used in modern computing, and modern microprocessors severely penalise the use of such techniques anyway [53] (that is, the use of self-modifying code and embedding data in code sections severely degrade performance). Note that this system does *not* preclude dynamic code-generation — some component can dynamically generate new component types (providing the new types are passed through the code-scanner).

However, a SISR-based system can still allow self-modifying code if required as long as such components execute while the processor is in user mode. Calls to the ORB will need to be made via traps, and not inter-segment branches as with code-scanned components. In such a scenario,

some 'sandbox' component would attach to the relevant trap vector, and forward calls to the ORB.

5.7.2 Variable-Length Instruction Sets

While most modern CPU architectures are RISC and so have fixed-length instructions, older designs (such as Intel x86) have variable-length instruction sets. Variable-length instructions complicate code scanning because immediate data can be executed as if they were instructions. For example, consider the Intel x86 code:

```
mov al, 0xFA
jmp -2
```

The first instruction places `FAh` into general-purpose-register `al`, and the second jumps back to the `mov` instruction, repeating indefinitely (the first instruction consumes two bytes, hence `-2`). This will cause the executing thread to live-lock, but will not affect the system as a whole. Now take the code:

```
mov al, 0xFA
jmp -1
```

This branches into the middle of the `mov` instruction, and attempts to *execute* the immediate data `FAh` as if it were an instruction. `FAh` is the opcode to disable interrupts on Intel x86 processors so this sequence of instructions will live-lock the *entire system.* In fact, since the branches in the above code snippet are immediate, code scanning could detect this problem by simply stepping through the code. However, if the branches are *indirect* then it is more difficult for the scanner to determine the branch-target before run time.

The problem can be overcome by having a tool insert a few extra instructions into code at compile time. These extra instructions force all indirect-branch targets to be aligned to some multiple of bytes. The code scanner then only allows indirect branches if they are preceded by some mask, forcing alignment. For example, the x86 instruction `ret` (which returns from a subroutine) is required to be preceded **immediately** by an instruction forcing the function to return to an 8-byte aligned address, such as "`and byte [ esp ], 0xF8`". This means that `ret`s are only valid if they are part of the code snippet:

```
and     byte    [ esp ], 0xF8    ; zero bottom 3 bits of return address ...
ret                              ; ... forcing returns to be 8-byte aligned
```

Note that while indirect branches are forced to target 8-byte aligned addresses, they themselves must not lie on 8-byte aligned boundaries. This way, the alignment mechanism protects itself. This is because if control is able to jump in between the mask and its associated indirect branch then the mask is not executed and the second indirect branch is not forced to be 8-byte aligned. Note also that this mechanism assumes that each thread has its own stack segment, so there is no race condition allowing concurrent threads to change the return address between it being masked and used. This "Time-Of-Check-To-Time-Of-Use" race can be avoided on systems where threads share stack-segments by forcing function returns to pop the return address into a register. The lower bits are cleared in the register and the return is a branch indirected through said register.

The algorithm for scanning using this mechanism is as follows. The flow of control is *simulated*, scanning for privileged instructions. *Simulation* in this context means that scanning starts at the first instruction and steps through until termination. One simulation-pass is started at each alignment interval (that is, a simulation-pass is performed starting at each 8 byte offset within the code). During the simulation-pass unconditional branches are followed, and conditional branches cause the simulation to fork: one simulation-pass executing as though the branch is taken, and one as though it is not.

If implemented naively, the above algorithm would have a complexity of $O(n^2)$. However, the algorithm can be optimised so that a simulation can be terminated under a number of conditions. These conditions are:

- Discovery of an undefined instruction (or any instruction that would cause the processor to fault, even in supervisor mode).
- Any indirect branch.
- A call to the ORB's `return` method (see Section 5.4).
- Any time a simulation comes across an offset already scanned. That is, the scanner marks which instructions have been scanned, and simulation-passes are terminated when an offset is simulated that is already marked.

The last of these optimisations (terminating a simulation pass when a location has already been scanned) means that the algorithm's complexity drops to $O(n)$: in the worst case the scanner will check an instruction at every byte offset.

Note that this technique could still produce 'false-positives' in the code scanner since the scanner might find privileged instructions that will never actually be executed. However, this can

be overcome by inserting NOPs before the apparently faulty instruction.

This technique will produce slight code-bloat, although less than the code-bloat associated with the move from CISC to RISC instruction sets. A prototype post-processor tool has been developed, that inserts the masks, alignments and NOPs to allow user components to be passed by the code scanner. Early experiments have shown code bloat of 5–10%.

5.7.3 Code Scanning with Paged Memory Protection

So far it has been assumed that SISR will use segmentation memory-management hardware to effect protection. Indeed, segmented memory protection is an integral part of the model described in this thesis. However, the code-scanning technique itself does not *require* segmentation in order to be effective: code scanning and segmentation complement each other to give SISR. Segmentation is useful with code-scanning because address-space changes are made visible in static code through segment-register-load instructions. Code-scanning compliments segmentation because it allows relatively complicated, OS-specific rules to be implemented that mandate under what circumstances what segment registers can be loaded.

Code-scanning is difficult to implement on paged architectures since context-switches are not statically visible as are segment-register loads. However, assuming a page-table load requires an explicit instruction, code-scanning could be implemented on paged architectures by allowing page-table loads under certain circumstances. Specifically, the code-scanner would permit instructions that load a new page table providing that such an instruction is followed immediately by an unconditional branch to a TCB entry point. This would mean that while untrusted code could load a new page table, it could not do any damage since control would be transferred immediately to the TCB. Note that if the page table loaded did not map in the TCB's entry point, then the following branch-to-TCB would fault, ultimately transferring control to the TCB anyway.

There are two problems with this approach. Firstly, most paged architectures have no notion of 'execute permission' — any readable page can be executed. This means that components would be free to execute their data as if they were instructions, thus rendering the code scanning useless. To overcome this problem, one can use a technique similar to the one used to enable code-scanning on variable-length instruction architectures (see Section 5.7.2). For example, all data could be placed in the top half of the machine's virtual address space, and all code placed in the lower half (pages marked read-only). All immediate branches can be checked by the scanner to ensure that they only branch into the lower half of the address space. The scanner ensures that indirect

branches have a mask inserted before them to ensure that the branch-target is in the lower half of the address range. E.g. `jmp r0` must be preceded by `and r0, 0x7FFFFFF8`, and indirect branches cannot be on addresses that are multiples of 8.

The second problem with this approach is less easy to solve: it almost certainly isn't worth it! That is, once the lengths described above have been gone to in order to allow code-scanning on paged systems, most benefits of SISR have disappeared. Most obviously, the low spatial overheads have disappeared since components' data and text sections are required to be a multiple of the page size. Also, temporal performance would likely be considerably worse than with conventional systems due to a page-table switch on every call to, and exit from, the TCB. The constraints mean that the simplicity of SISR on segmented architectures is lost too.

In summary, code-scanning itself does not require segmented-memory-management hardware to work. However, segmentation is an integral part of the SISR model — code-scanning and segmentation compliment each other to provide SISR which is more than sum of its parts: code-scanning is much more than simply eliminating "expensive processor mode changes".

5.8 Summary

This chapter has outlined SISR, a novel protection model. The model enables true decomposition of the TCB through the elimination of separate processor modes. Protection is maintained by combining segmentation with 'code-scanning' (that is, scanning components' code and rejecting those components that contain instructions which they are not privileged to execute). The code-scanning allows loading a segment-register to be considered a privileged operation, so that placing a component in its own segment protects it from others. More precisely: holding a segment's selector in a segment-register becomes a capability to access that segment. Code-scanning compliments segmentation because it allows relatively complicated, OS-specific rules to be implemented that mandate under what circumstances what segment registers can be loaded. Segmentation compliments code-scanning because an address-space switch is performed through a segment-register load; an explicit instruction easily visible to the code scanner.

This new model will allow a context switch to be effected by three segment-register loads (code, data and stack), and so will result in a significant reduction in context-switching temporal overheads. However, because a segment-register load is a privileged operation, a component must call upon the Object Request Broker (ORB) in order to load segment-registers and effect a context switch on its behalf. This indirection is just one inter-segment branch and a data-

segment-register load, and so will increase a context switch to five segment register loads in any real implementation. Note that the omission of processor-mode switches is a relatively small aspect of SISR's performance advantage — most of the advantage comes from requiring just a few segment-register loads in order to effect a context switch. Furthermore, there are many advantages other than reductions in protection and context-switching overheads including simplicity, improved decomposition and flexibility.

A suitable implementation of this model is likely to meet the requirements set out in Section 2.3. The low context-switching overheads and use of segmentation to effect memory protection will allow protection at a fine granularity while maintaining good performance. Together with the abolition of the system-call barrier, this will in turn lead to improved software-engineering (at both the system and application levels).

Chapter 6

Go! and GTE — A Proof Of Concept SISR Implementation

6.1 Introduction

In order to prove the concept of SISR, a prototype ORB — called Go! — has been implemented. Go! runs natively on the Intel 80386 and above (known as IA32) [52]. The Go! ORB must meet the goals set out in Section 2.3. Although Go! is a prototype, it is intended to be a complete system, suitable for use in real-world applications. Note that if Go! were to be used in a real-world application it is likely that several changes would be required. However, no "short-cuts" have been taken in Go!'s development.

Go! is not an operating system in the conventional sense, but rather a component architecture and infrastructure which is suitable for the construction of (amongst other things) operating system services. To provide a useful proof-of-concept, it is important to show that traditional OS services can be implemented in this framework. To this end, a collection of components has been developed that runs on top of the Go! ORB to provide a library operating system. This library OS is called GTE (the Go! Test Environment). Unlike Go!, GTE is purely an experimental system — that is, it is not intended that GTE offer a system complete enough for real-world use, just to prove that Go! is suitable as a base for operating system construction.

This chapter describes the architectures of both Go! and GTE, and then Go!'s interface and implementation in more detail.

6.2 Architecture Overview

Go! provides an architecture quite different from that of traditional operating systems. A kernel and applications are replaced by a collection of components, some of which offer OS services, and others application ones. This architecture can be thought of as a 'zero-kernel' — there is no longer a kernel, just components.

6.2.1 Components: Types, Methods and Instances

All code and data in Go! are encapsulated into components. Each component is an *instance* of some *type*. Every datum belongs to exactly one instance, and every instruction to exactly one type[1]. There may be many, one or no instances of a given type at any time. Note that the terms 'component' and 'instance' can be used interchangeably.

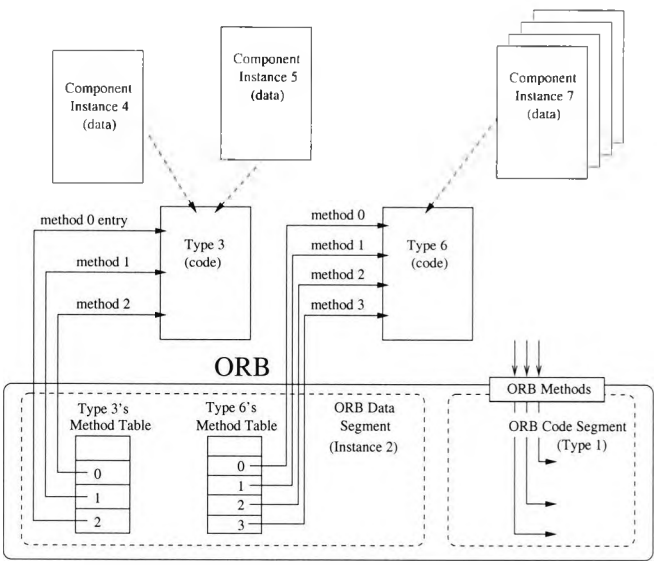


Figure 6.1: Types and instances of Go! components

Figure 6.1 shows an example system with the ORB, two instances of type 3 (instances 4 & 5), and many instances of type 6 (starting from instance 7). The ORB is instance 2, and of type 1. Note that types and instances share a common namespace. Note also that in this example base types and initial data segments are ignored in the interests of simplicity.

Component types export to the ORB a set of *methods*. These represent specific entry points within that type's code segment. A component may invoke the service of another by requesting

[1]Sharing data between components is facilitated through granting a type access to another type's data. See the description of the `null` base type in Section 6.3.3 for details.

that the ORB invoke a given method on its behalf (known as a 'method call'). A type's methods are numbered (starting from zero), so "to call method number 3" on a component means to call the fourth method enumerated on that component's type. Note that in Figure 6.1 type 3 has three methods, and type 6 has four.

Each component has two extra methods, not enumerated in the normal way: a constructor and a destructor. The constructor is called by the ORB when an instance is created. The instance does not appear in the namespace until the constructor has completed (thus no other methods will be invoked until after the constructor has returned). Similarly, the destructor is called on component destruction. The component instance disappears immediately before the destructor is called: no methods will be called on an instance once its destructor has been invoked.

6.2.2 ORB Methods

Like all components, the ORB's services are invoked through calling one of its methods. The most interesting ORB methods are `call` which invokes a given method on a given component, and `return` which returns from the most recent `call`. The ORB also has methods to `create` and `destroy` components, as well as to `install` and `uninstall` implementation types. Several other ORB methods are necessary for a complete implementation; a full list is given in Section 6.4.

Since it is necessary to invoke an ORB method (`call`) in order to invoke methods on other components, the ORB is necessarily invoked differently (otherwise infinite recursion would result!). As outlined in Section 5.4, other components call ORB methods by issuing an immediate inter-segment call. A Go! ORB method is invoked by the instruction `call 8:8`n, where n represents the ORB's method number to call.

Only certain ORB methods are safe to be called by untrusted components. These are those that control inter-component communication (such as `call` and `return`) and read-only methods (such as `get_type`). All other ORB methods (such as `install` or `destroy`) are known as *privileged methods* — these methods are intended for use only by the library operating system (and other trusted components). The code scanner is able easily to detect exactly which ORB methods are invoked by each component type: if the type's code segment contains the instruction `call 8:8`n then it is assumed to invoke ORB method number n. The code scanner is thus able to restrict which ORB methods are available to untrusted components — that is, the code-scanner ensures that untrusted components may contain inter-segment branches only immediately to *safe* ORB methods.

Because certain ORB methods can be called safely only by trusted components, in the interests of efficiency the ORB need not perform sanity checks on the parameters passed to these privileged methods. That is, if invalid parameters (such as an invalid reference) are passed to privileged methods the ORB's behaviour is undefined[2].

Safe ORB methods are analogous to UNIX system calls: control is transferred atomically and securely from an untrusted to a trusted entity.

6.2.3 Inter-Component Communication

As already explained, components communicate with one another by invoking methods via the ORB. Go! uses thread-tunnelling for inter-component communication. Thread-tunnelling was chosen over message-passing for the following reasons:

- If fine-grained decomposition is to incur low overheads, then components themselves must be lightweight. The association of at least one thread of control with each component will be detrimental to this cause: it is anticipated that most finely-grained decomposed systems will have many more components than threads.

- The most common means of cross-protection-domain communication at the programmer's level is RPC. Thread-tunnelling is well suited to this. While message-passing is better suited to asynchronous communications, it is not anticipated that asynchronous communications will play a big role in component-based systems. In [101], Andrew Tannenbaum writes "...in Amoeba 2.0 we made a truly dreadful decision to have asynchronous RPC...Our advice to future designers is to avoid asynchronous messages like the plague."

- In a finely-grained, synchronous, decomposed system, the latency of inter-component communication is likely to present the biggest performance bottleneck. Thread-tunnelling offers very low latency.

- Go! is intended to be suitable for use in real-time systems. Real-time systems typically require low, but more importantly, *predictable* and *bounded* latency. This is simpler with thread-tunnelling. In addition, thread-tunnelling can be used to eliminate QoS crosstalk (see Section 4.5). This is because a single thread is used to carry out one operation, even if that operation is performed by different domains. That is, the same scheduling attributes remain associated with a given operation as its thread crosses protection domains.

[2]However, the ORB is free to perform sanity-checks on certain parameters and throw exceptions should these checks fail (for example, the current implementation does so only when built with the `_DEBUG` macro defined).

In fact, Go! takes thread-tunnelling even further. Not only does the operating system support thread-migration across components, but fully-fledged RPC is promoted to the principle OS communications primitive. That is to say, RPC is not implemented in language libraries (as is normally the case), but directly through an ORB method. This is made possible because Go! promotes the status of a procedure to a fundamental OS primitive (through component types' methods).

6.2.4 Less is More: the Zero-Kernel Approach

To allow for simple architectures, the Go! ORB is designed to be the 'lowest layer' of software in any system. That is, the ORB does not rely on the services of any other components. This allows a strict separation of the component infrastructure and the operating system.

Go! defines a component architecture and provides a basic infrastructure through an ORB and a small set of base component types (the `null` type, a `stack` type and a few exception types — see Section 6.3.3). System functionality such as interrupt handling or memory management is provided by components that operate within this framework.

Note that the statement that system services such as interrupt handling are implemented by components does not mean that interrupts are farmed out to user level via veneers as with exokernel-like systems. The ORB remains blissfully unaware of interrupts, paging and even code-scanning. Operating system services such as interrupt handling and memory management can be implemented by components that have no architectural difference from components that implement user-level services such as spell-checking. Normal components are able to implement system-level services because all code executes while the processor is in kernel mode (see Section 5.2.2).

This architecture of a simple ORB with all OS services delegated to components facilitates the construction of an operating system that is *truly* component based.

6.2.5 The Go! Component Model and Object Orientation

Over the last decade, object-orientation has become the software-engineering tool of choice. It is therefore desirable that the Go! component-model can be viewed as object-oriented. There are two aspects to Go!'s component-model: *the programmer's view*, and *the implementation.*

High-level programming abstractions like inheritance (and to some extent interfaces) are not supported directly by Go!. Instead, the ORB implements only rudimentary support for object-

orientation, on top of which the programming environment (specifically the "IDL-compiler"[3]) can create the illusion of a fully-featured, object-oriented system.

Most obvious is the lack of specific support for interfaces in the traditional sense. An interface in Go! is simply a set of *methods*, which are just entry-points into a component's code segment. Go! provides no facilities to match or type-check interfaces. However, such support can be added easily by the library operating system. For example, an interface management component might record the interface that each implementation realises, and provide services to locate a component instance based on interface.

There is also no explicit support in the ORB for an implementation with more than one interface. However, the library operating system can provide support for this by overlaying different components at the same linear address. There is rudimentary support for the programming-environment to present an object-oriented view where different levels of the inherited interface can be realised by different implementations (known as implementation inheritance). This support takes the form of the ability to allocate several consecutive object references at component construction. Refer to Appendix C for a complete description of how this works.

Providing only *support* for object orientation (rather than full object orientation) at the ORB level improves both performance and flexibility. Performance because the ORB is not weighed down by expensive and complicated features such as multiple-inheritance, and flexibility because the exact component-model used can be specified by the programmer. For example, two programmers might access the same service differently: one using an object-oriented IDL compiler that generates C++ header files, and one using a procedural IDL compiler that generates C headers.

6.3 Go! and GTE Architecture in Detail

This section presents Go! and GTE's architectures in more detail. This includes a description of the different inter-component communication primitives, software exception support and the Go! base types. An overview of OS service presentation and GTE is also given.

6.3.1 RPC Variations and Optimisations

Go! offers several inter-component communication primitives, all based on RPC call and return semantics. The different 'flavours' of `call` offered by Go! are discussed in this section. Briefly

[3]An IDL compiler is a tool that takes as its input a specification of a component's interface in some 'Interface Definition Language' (IDL) and outputs 'skeleton code' and 'header files' which aid the programmer significantly.

there are three types of call offering different trade-offs between latency, protection and setup overheads. There is also a non-returning variant — Go!'s analogue of a `goto`.

It is important that method implementations are not required to be aware which variant of `call` was used to call them. To facilitate this, despite the ORB providing 4 primitives with which to call a method, there is just one primitive with which to return: `return` is able to determine which variant of `call` was last issued, and return appropriately.

`f-call`

As shown in Section 7.2.1, timings for `call` have shown impressive latency. However, this is considerably worse than the optimum performance. Based on instruction latencies documented in [53], the theoretical performance of a null-RPC using Go!'s `call` primitive is 83 cycles. The measured latency is somewhat worse than this at 252 cycles. This sub-section starts with an investigation as to the reasons for this discrepancy, and then presents Go!'s solution.

Experiments have shown that the discrepancy in predicted and measured performance is due to a processor stall caused by writing to the Pentium's segment-descriptor table. Initial investigations have demonstrated that such a write stalls the processor for approximately 50–60 cycles. Although the exact reason for this stall is not known, it is likely due to a flush of the processor's internal state. Note that a write to the segment-descriptor table does not affect the normal operation of the processor until a segment register is next loaded, and so one would not expect such writes to the descriptor table to be 'special'. However, IA32 executes several segment-register loads in order to service an interrupt or fault. It also requires that faults are delivered immediately after the instruction that triggered them (know as 'precise exceptions' [46]). Since the instruction immediately following a write to the descriptor table might fault, attention must be paid to the synchronisation of the delivery of this fault (and the segment-register loads this implies), with the new descriptor table. Since most systems do not alter segments' sizes dynamically, it is reasonable that the Intel engineers paid little attention to optimising this case. That is, it is unlikely that these stalls are intrinsic to segment-based systems, or even to the IA32 design; merely that Intel did not spend silicon budgets or design-time optimising this unusual case. Unfortunately, a round-trip RPC on Go! requires two writes to the descriptor table: one on the `call` to shrink the stack, and one on the `return` to re-grow it.

In the case that particularly low-latency RPC is required, the `f-call` (fast-call) primitive can

be used. `f-call` requires the invoking thread has several stacks in a doubly-linked list[4]. Stacks are pre-created at suitable sizes, and switched to the next in the list on an `f-call` and switched back to the previous one on a `return`. The stacks overlap one another so that parameters can be passed on the stack as normal.

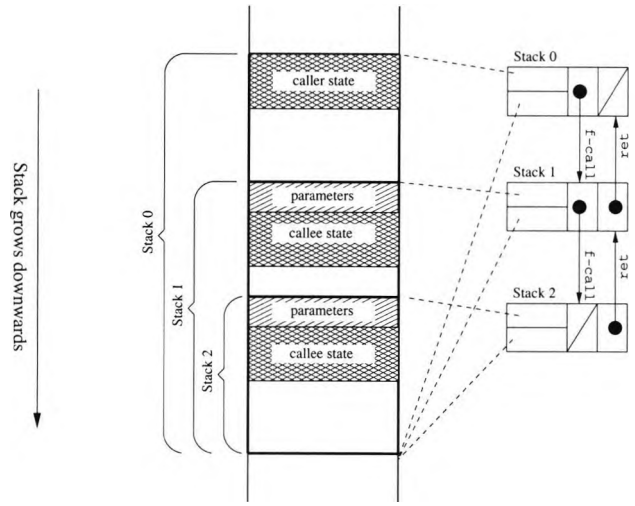


Figure 6.2: Overlapping `f-call` stacks

Figure 6.2 shows three stacks set up for use with the `f-call` primitive. Here, execution starts with stack 0 active, linked to stacks 1 and 2, as shown. If an `f-call` is issued, the active stack will be switched from stack 0 to stack 1 (the next in the list). To enforce protection, the caller must ensure that, at the time the `f-call` is issued, all its stack-based state is stored in the region of stack 0 that does not overlap stacks 1 or 2 (that is, the top-most region). Parameters must be passed starting from the top of stack 1 so that the callee may access them. Note that this requires the caller know where stack 1 starts within stack 0. This should not be a problem because it is anticipated that the caller will create stack 1 itself before issuing the `f-call` (probably indirecting via some memory-management component). Note that stack 1 will probably be created just once (possibly during the caller's initialisation) since `f-call` would *not* offer low latency if each `f-call` was associated with a stack creation!

A second `f-call` can be issued with stacks as in Figure 6.2, which would switch from stack 1 to stack 2 in exactly the same way stack 0 was switched for stack 1. If the callee then issues a `return` stacks will be switch back to stack 1, and finally to stack 0 on a subsequent `return`.

Note that the stacks used with `f-call` are in no way special — any two stacks can be linked

[4]Each stack may exist in only one `f-call` 'stack chain' at a time.

to together in an `f-call` chain. Conventional `call`s (or `t-call` or `xfer` — see subsequent subsections) can be safely issued from methods called with `f-call`. For example, with the layout shown in Figure 6.2 one might issue the following sequence: `f-call`; `call`; `f-call`; `return`; `return`; `return`. Here, stacks would be switched on the first `f-call` to stack 1, and then on the `call` stack 1 would be shrunk. The second `f-call` would switch to stack 2. A subsequent `return` would switch back to the shrunk stack 1. The next `return` would re-grow stack 1 to its original size, and the final `return` would switch back to stack 0.

Note that pre-creating stacks means that the caller must know in advance how much stack space will be required for each call. This is no different from stack creation in conventional systems however since only *at least* enough stack space must be provided. Note that if paging is enabled the stacks can be backed by physical memory only as required.

Since each `f-call` requires a new, pre-created stack, and the standard `call` primitive offers quite reasonable RPC latency, it is anticipated that `f-call` will be used only where RPC latency is critical (such as within a tight loop, or when under stringent real-time constraints).

`t-call`

Often a client trusts its server. For example, a user application is likely to trust the file server and GUI implicitly. In such cases it is not necessary to protect the client's stack frame during RPC (although, by the nature of the Go! component-model, the client's *state* will be protected since only one data segment can be active at a time). This means that if the client is prepared to trust the server not to corrupt its stack frame, `t-call` (trusting-call) can be used to give the speed advantages of `f-call` without the penalty of requiring another stack. In short, programmers can choose any two from low RPC latency, low/zero setup overheads (that is, no stack creation), and protection.

Note that `t-call` requires that the issuing client trust the server, but not that the server trusts the client. It would also be unacceptable if other components needed to trust a server called with `t-call`. Hence the ORB guarantees that the server can return only to the client that issued the `t-call`. Otherwise, if client A trusts server B and issues an `f-call`, even though a third party C does *not* trust B, B would still be able to return into an arbitrary location of C. When one considers that the same programmer might develop client A and server B, the security requirement that `t-call` is guaranteed to return only to its caller is clear. `t-call` is also guaranteed to return to the relevant instruction within the caller, useful when clients do not *trust* the server, but can

afford to 'loose' the calling thread.

`xfer`

A client does not always require that control be returned to it after an RPC. For example, destroying the current thread or servicing an interrupt will not require control to return. A more common example however is when the last thing that a method does before returning is to call a method on another component (`return` is defined to return control to the last `call`, `f-call`, or `t-call`). This situation is shown in the C code:

```
int
foo() {
    /* .
       .
    do some processing here
       .
       .  */

    return bar();
}
```

In this situation the function `foo` could issue an `xfer` to `bar` knowing that when `bar` eventually returns it will return directly to `foo`'s caller. This is illustrated in figure 6.3.

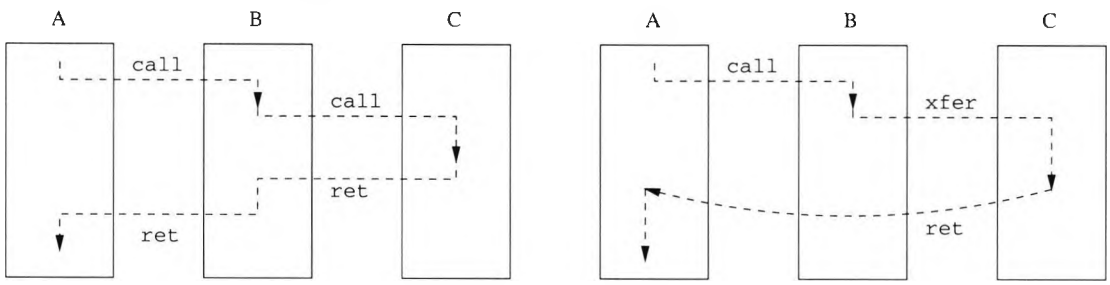


Figure 6.3: Returning from a previous `xfer`

This technique can save several useful cycles if used as above, and can lead to significant performance improvements if used repeatedly, such as with tail-recursion optimisation [97]. It is also *necessary* for the implementation of certain algorithms if stack overflow is to be avoided (that is, where a programmer would traditionally require a `goto` rather than a function call[5]).

[5]Despite 'text-book-teaching' [28], if used sensibly, `goto` can be a useful programming primitive [64, 90].

6.3.2 Software Exceptions

Often control flow must indicate the occurrence of erroneous or exceptional circumstances. The desired behaviour for such control often differs from the standard call and return semantics seen so far. This section details Go!'s support for software exceptions and gives examples of their use.

Software-Exception Support

Most modern software engineering practises make use of *exceptions* to have a server signal its inability to continue execution to a client [91]. However, most operating systems provide little support for this, instead relying on software layers above (usually the high-level language) to implement exceptions if required. This technique makes it difficult for exceptions to propagate across modules implemented in different languages (or even different compilers), and modules are usually restricted to signalling a server's inability to continue via integer error codes. Also, because Go! provides RPC primitives, it is necessary that Go! provide at least some way to unwind the call stack if exception support is to be enabled. It is also desirable that components are able to *intercept* certain exceptions. Such ability would allow efficient and elegant software solutions impossible in other systems. For example, the Go! ORB provides no support for distribution. Such behaviour could be added by other components either through the use of proxies, or by having some locator service intercept all exceptions as a result of using invalid component references (see Section 8.3.1). Similarly, the ORB has no direct support for paging. Instead, a `pager` component might intercept all 'out-of-memory' and 'page-not-present' exceptions to provide demand paging.

Exception Implementation

If a server wishes to throw an exception to a client, it must create a component to represent that exception, and invoke the ORB's `throw` primitive (see Appendix A.3.1). This is used to throw an exception to the most recent caller. Its behaviour is similar to that of the `return` primitive, except that once the stack is unwound, rather than transferring control to the instruction following the most recent `call`, a software interrupt is triggered (on vector 30h). Some exception manager component can then attach to this interrupt vector (probably via some interrupt-handling component). In response to receipt of such interrupts, the exception manager will most likely `xfer` to the caller. However, if the exception manager realises the interception of exceptions, it might pass certain exceptions on to interested third parties.

Throwing software exceptions via interrupts means that not only can exceptions be intercepted,

but that the same mechanism can be used to deal with software exceptions and hardware exceptions such as page-faults or divide-by-zero errors. It also means that the ORB can throw exceptions without being aware of what component is responsible for exception management.

Preemption

Go!'s thread-tunnelling RPC model means that it is sometimes necessary to preempt a thread from a server. This is because when a client requests some service, the client's thread is migrated to the server. If the server then times out (for example, through entering an infinite loop), the client must have some way to 'reclaim' that thread. This problem is non-trivial, because it is not possible to tell whether the server is in an infinite loop or if the operation will complete given enough time [105], and because the server may have claimed resources that need releasing.

Exactly how this is done is not defined by Go!, but left to the library operating system. One solution is to set up a time-out which expires on receipt of a timer interrupt[6]. If the server has not completed by receipt of the time-out, an exception is delivered to it. This exception is known as a 'two-minute-warning', which gives the server some period to complete its operation (likely to be of the order of milliseconds). Failure to respond to the 'two-minute-warning' exception in time will result in control being revoked from the server. How this is done will depend on the library-OS policy; one such policy is to terminate the server without prejudice; another to have the offending thread preempted, and an exception thrown to the client.

Another solution is for the caller to arrange for a new thread to be invoked in response to some time-out. If the server responds in time, the time-out is cancelled. If the server does not respond, the client continues execution with the time-out thread. If the caller detects the return of the original thread after the time-out, it must take appropriate action (such as terminating the timed-out thread, or using it to throw an exception).

The Call Chain

As execution continues and threads tunnel through several components, a 'call-chain' is built up. Assuming correct operation, given time this call-chain will unwind. However, it cannot be guaranteed that the caller component will still exist when the callee returns control (due to the caller's normal or erroneous termination). This is a problem that needs solving in any thread-tunnelling RPC system. Go!'s solution to this is to throw an exception if control is returned to a

[6] How exactly interrupts are handled is defined by the library operating system. For the purposes of this discussion it is assumed that there is some collection of components capable of receiving interrupts, suspending the current thread, and scheduling a new one (see Appendix B.6 for an example implementation of such a mechanism).

non-existent component. Recall that the final recipient of an exception is defined by the library operating system — however, it is anticipated that most library operating systems will throw the exception to the caller's caller. See Figure 6.4 for an illustration of this.

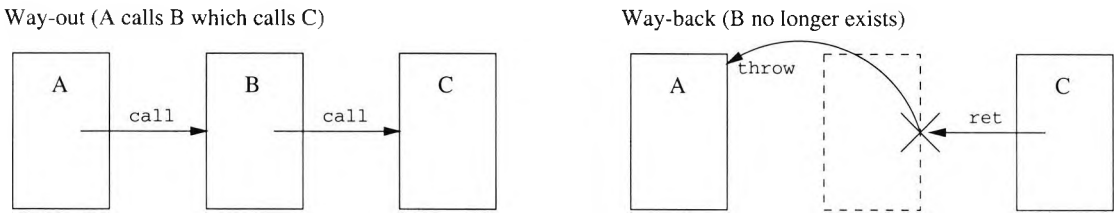


Figure 6.4: Unwinding a broken call chain

This is implemented by having components enter a special 'zombie' state between their destruction and any threads returning. A zombie component is not really a component at all — in reality a record is kept of the zombie's reference so that it may not be re-used. The zombie's reference is stored so that any attempted return to a zombie is caught via a hardware-protection fault (see Section 6.5.1 for details). This is possible because the ORB keeps a `call_count` for each component (that is, how many threads have entered a component, but are yet to leave via a `return`, `throw` or `xfer` — see Section 6.3.1).

6.3.3 Base Component Types

As well as the ORB, Go! defines several 'base' component types. These base types are:

`null` A 'memory object'. The null type contains no behaviour, but instances of it may have state. That is, it is a data segment with no associated code segment or initial data segment. The `null` type is necessary because the ORB requires that new types are presented to it as a `null` component with its data in the appropriate format. It is also the primary means for sharing data between components. To share data, an instance of the `null` type can be created that overlaps another component's data. Data are shared by placing the `null` component's data segment selector in the IA32's `es` segment register.

`stack` Similar to the null-component, except that the component can be used as a stack by `switch_stack`. This is necessary for the ORB's rudimentary support for multithreading (see Section 6.3.4 and Appendix A.5 for more details). Stack components can also be linked to form a stack-chain for use with `f-call` (see Section 6.3.1).

`xcp` The `xcp` interface is used for the ORB to signal various error conditions. This is necessary since the exception mechanism requires that the error is identified by a component. The `xcp` type has one method: `uint info()`, which returns an integer giving further details of the cause of the exception.

In reality, the `xcp` type is just a (notional) interface which is shared by seven implementation types. These are:

`xcp_orb_invalid` An instance of this type is thrown via an exception in response to an attempt to invoke any ORB method with invalid parameters (for example, calling a method that doesn't exist, or attempting to install a new type using an invalid data format (see Appendix A.2)). `info` returns an ORB-implementation-specific value for use with debugging.

`xcp_orb_noref` An instance of this type is thrown via an exception in response to an attempt to invoke a method on a component using an invalid reference. `info` returns the faulting component reference. This exception can be intercepted by the library operating system to aid distribution (see Section 8.3.1 for details).

`xcp_orb_nomem` An instance of this type is thrown via an exception if the ORB requires more linear space in order to continue. Any component servicing this exception should respond by calling the ORB's `linear` method, passing a region of linear memory for the ORB's use. `info` returns the amount of memory by which the ORB is short, or zero if the exception is as a result of an attempt to create a component that overlaps the ORB's internal data structures (see Section 6.3.4) — in effect, when `info` returns zero this indicates that linear memory has been exhausted. Note that the ORB requires some dynamic memory in order to create exceptions, meaning dead-lock can occur (that is, the ORB needs to throw an `xcp_orb_nomem` exception to state that it needs more memory, but it cannot create such an exception because its memory is exhausted). To avoid this dead-lock, an `xcp_orb_nomem` component instance is pre-created and associated with each `stack` when it is created.

`xcp_orb_nostack` An instance of this type is thrown via an exception if there is an attempt to `f-call` from the final element in a stack-chain. `info` returns an undefined result. This exception can be intercepted by the library operating system to provide demand-allocated stack-chains for use with `f-call`.

`xcp_orb_noret` An instance of this type is thrown via an exception if there is an attempt to `return` from a stack with no `returns` pending — that is, an attempt to return off the top of the call stack. For example, if the first method called on a new thread attempts to return the ORB will throw an `xcp_orb_noret` exception. `info` returns an undefined result.

`xcp_orb_zombie` An instance of this type is thrown via an exception if there is an attempt to `return` to a zombie (see Section 6.3.2).

`xcp_orb_gdtfull` An exception of this type is thrown when a slot is required in the segment-descriptor table (known as the 'GDT' on IA32) but none are available (due to the GDT being full of ObjRefs which are locked in — See Section 6.5.1). This situation is analogous to a demand-paged virtual memory system not being able to reject a page from physical memory because all pages are locked down. That is, this should never happen in normal operation, but it is better to throw some exception rather than have the system just lock up. `info` returns an undefined result.

6.3.4 Presentation of OS Services

Traditional operating system services such as memory management, interrupt dispatching, and even code-scanning are provided by different components running on top of the ORB. This section describes first the interaction between the Go! ORB and library OS components in the general case, before describing an example library OS implementation: GTE.

Decomposition of Memory Management

In order to support dynamic creation and destruction of components (and dynamic installation/uninstallation of types) memory must be managed dynamically. However, in the name of decomposition, the ORB requires that some external component manage linear space. This is achieved by requesting that a client specify not only a component's type on creation, but also its linear address. It is anticipated that most clients will not care where a component is placed in linear memory, and indeed allowing clients to specify their children's linear address is a security flaw. In protected systems, clients will be forced to create other components indirectly via some component that manages linear space. This is shown graphically in Figure 6.5.

Such indirection is forced because the code-scanner considers the ORB's `create` and `destroy` methods as privileged (see Section 6.2.2). This means that untrusted components are prevented

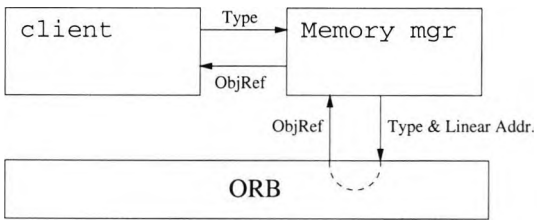


Figure 6.5: Component creation via a memory manager

from specifying their children's linear addresses.

Decomposition of Code-Scanning

So far it has been implied that code-scanning is performed by the ORB as part of component-type installation. However, since code-scanning is a fundamentally different task than component-management, the code-scanner is a separate component in order to minimise coupling. In other words, code-scanning is left to the library operating system constructed on top of the ORB. To ensure protection, all component type installation must be forced to go via the code-scanner (otherwise untrusted components could directly install new types that contain privileged instructions). Thus the ORB's `install` method is privileged. This means that the code scanner will identify calls to the ORB's `install` method and so prevent components from installing new types directly. That is, the code-scanner refuses to allow types to be installed that install other types directly, forcing installation to go via the scanner. Thus code-scanning can be used to enforce that code-scanning is performed!

Note how the indirection of component installation via some privileged scanner component mirrors the indirection of instance creation via some memory-management component.

Multithreading in Go!

The Go! ORB does not provide multithreading, but is designed so that multithreading can be provided easily by a library operating system. Multithreading support is provided through the ORB being re-entrant. To achieve this it is necessary that the ORB support multiple concurrent stacks and be able to switch between them on a thread switch (since the ORB manipulates stacks in order to provide protected RPC). It is anticipated that with each thread the library OS will associate one stack (or more specifically, one stack-chain — see Section 6.3.1). All ORB call-related state is stack specific: the ORB maintains a separate copy of each stack-specific variable. All other ORB state is manipulated in such a way that the ORB is completely preemptable (that

is, non-blocking, lock-free synchronisation [40] is used throughout the ORB).

Up until now, `return` has been defined as returning from the most-recent `call`. These semantics are extended so that `return` will return from the most-recent `call` *of the current stack.* Since each thread has its own stack, this means that the ORB's `call` and `return` primitives are thread-safe[7].

The exact architecture of multithreading is — naturally — defined by the library operating system. Most likely some scheduler component will switch threads by responding to a timer interrupt, saving the currently running thread away and scheduling some new one. See Section 6.3.5 for an example implementation.

More details are given in Section 6.3.5.

6.3.5 GTE: An Experimental Library OS for Go!

To provide a complete proof-of-concept of the protection model presented in Chapter 5, a set of components has been developed for Go! that provides a 'toy' library operating system. Ten components cooperate to provide interrupt handling, multithreading, error handling, protection, device drivers and applications. Although minimal, GTE is complete enough to show that it would be possible to develop a more complete system on top of Go! (for example, a POSIX [39] compliant one). This set of components has been called GTE (Go! Test Environment), and is made up of the component types shown in Table 6.1.

`comp_lib`	Component library and GTE boot-strapper
`mem_mgr`	A simple linear-memory manager
`idisp`	An interrupt dispatcher
`sched`	A preemptive, round-robin, single-priority scheduler
`scanner`	A simple code-scanner
`xcp_mgr`	An exception manager
`thread`	A component to represent a thread
`video`	A console driver
`keyb`	An interrupt-driven keyboard driver
`cli`	A basic command-line interface (that is, a simple shell)

Table 6.1: The component types comprising the Go! Test Environment

Note that the `video`, `keyb` and `cli` components are not so much part of the library OS, rather they represent 2 device drivers and a small application. The interface, design and implementation of each of these components is described in detail in Appendix B.

[7] Multiprogramming models that do not associate a stack with each thread of control (such as with continuations [29]) cannot maintain procedure-call state between context switches, and so will still be 'thread-safe' with this model.

There follows a brief description of the three GTE components most interesting from a decomposed OS point of view: the memory manager, interrupt dispatcher and scheduler.

Memory Management in GTE

The GTE `mem_mgr` component provides memory management. Since GTE is a proof-of-concept system only, the `mem_mgr` is very simple, using a free-list implementation to manage free and allocated linear memory. The memory manager exports four methods: `install`, `create`, `destroy` and `uninstall`, which correspond to the ORB methods of the same name. The memory manager allocates linear space on a `create` and releases it on the corresponding `destroy`. Only types installed with the `mem_mgr`'s `install` method may be created through its `create` method.

The `install` method forwards an installation request to the Go! ORB only after a successful pass by the `scanner` component, which implements the code scanning mechanism described in 5.7.2. If the component calling `install` is known by `mem_mgr` to be one of the core GTE components then privileged instructions are permitted (although the scan is still performed in order to properly manage I/O port allocation). This way OS components can contain the privileged instructions necessary, including calls to privileged ORB methods. The `mem_mgr` can determine easily whether `install` was invoked by a core GTE component or an unknown (so untrusted) one using the ORB's simple authentication mechanism (see Section 5.4.1).

Interrupt Management in GTE

Along with `sched` (the scheduler), `idisp` is the most interesting component from a decomposed-OS point of view [67]. `idisp` demonstrates nicely the decomposition of OS kernel services made possible by SISR. The `idisp` component (interrupt dispatcher) is unique amongst operating systems: it is an application-level component that manages interrupts *in their entirety.* `idisp` contains an interrupt-vector table in its data segment, and points the CPU's interrupt mechanism at it. Because of Go!'s simple architecture, no special arrangements need to be made for the interrupt dispatcher. Since `idisp` is a core GTE component, the scanner permits its use of the privileged instruction `lidt`. This is used by `idisp` to direct all interrupts to its own code segment. Because `lidt` is a privileged instruction, untrusted components cannot install themselves as an interrupt dispatcher.

Other components can register an interest in a particular interrupt by calling `idisp`'s `attach` method. The `attach` method takes a GTE thread to attach to a given interrupt vector. The thread

is normally blocked; it is unblocked (via the `sched` component) on receipt of the relevant interrupt. The current implementation allows only one thread per interrupt vector, issued on a first-come first-served basis. Protection is maintained by having components that receive interrupts installed early in the boot sequence.

Preemptive Multithreading in GTE

The `sched` component provides preemptive multithreading. It does this by receiving timer interrupts from `idisp` (via an `xfer`), in response to which the state of the currently active thread is saved, and a new thread scheduled. The scheduler operates with interrupts disabled to ensure that it is only called by the interrupt dispatcher when it is ready.

The implementation of `sched` is somewhat intricate, and its interface is intertwined with `idisp`'s. Note that despite the interfaces of `sched` and `idisp` being mutually dependent, the *implementations* are properly separated.

On construction, a 'scheduler-thread' is created, and `sched` attaches this thread to itself so that its `click` method will be called. The 'scheduler-thread' is then attached to the `idisp` on vector 32 — the timer interrupt. This means that the `click` method is called on each timer click (approximately 18.2 times per second). The `click` method does nothing except `xfer` to the `idisp`'s `intr_done` method, which in turn `xfer`s to `sched`'s `block` method which chooses a new thread. The result is that each clock results in the currently executing thread being suspended, and a new one being scheduled. Therefore in response to a time interrupt: (1) the current thread is suspended; (2) the scheduler-thread is woken; (3) the scheduler-thread is suspended; (4) the next thread is resumed. This is shown in more detail in Figure 6.6.

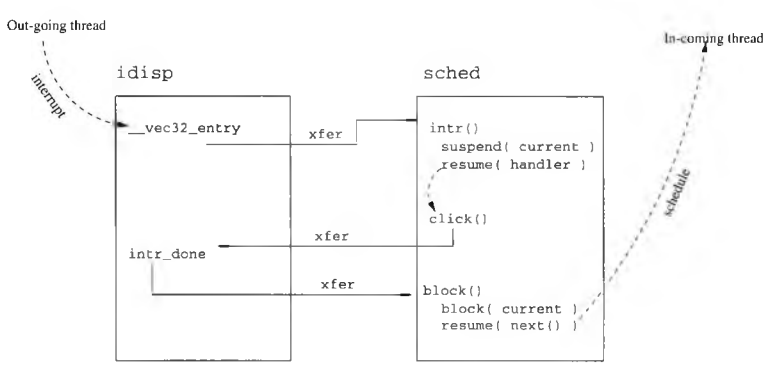


Figure 6.6: `sched` and `idisp` interaction

This mechanism results in extra temporal overhead due to the unnecessary resumption of the

intermediate 'scheduler-thread' between thread switches. However, this mechanism means that the timer interrupt is handled just as any other interrupt. Furthermore, in a more complete system it is likely that `sched`'s `click` method will perform useful work, such as checking for time-outs.

6.3.6 Summary of Section

GTE has done more than show that the Go! ORB is a suitable base for the construction of library operating systems. Its development demonstrated advantages in systems-software engineering — the system is decomposed into separate components giving benefits associated with modularity [76]. Most notably, during GTE's evolution, modifications to a component's behaviour were well isolated (that is, changes did not propagate throughout the system). Also, protection faults can be easily traced back to the offending component — the fine granularity of protection means the scope of the bug's source is reduced, dramatically easing debugging.

Note that apart from a simple greeting message being displayed on start up, the Go! ORB has no requirement that it be run on the IBM PC (although an IA32 processor is presumed). GTE however is required to be run on an IBM PC, with components such as `video`, `idisp` and `keyb` presuming PC-specific features. This is more evidence of the extent to which Go! is decomposed — even in the final binary images, which software components require the presence of which hardware components is clearly separated. That is, the ORB, `sched`, `thread`, `comp_lib`, `scanner`, `xcp_mgr` and `cli` components are platform independent (although tied to IA32), while `mem_mgr`, `idisp`, `video` and `keyb` require the presence of an IBM-compatible PC.

6.4 ORB Interface Overview

The ORB exports several methods used to manage components. Specifically, these methods are:

`ctor` called by the multiboot [36] compliant boot-loader at boot time

`call` call a given method on a given component (four 'flavours'):

- `call` call a given method on a given component, protecting the stack
- `f-call` 'fast-call' a given method on a given component, switching stacks
- `t-call` 'trusted-call' a given method on a given component without protecting the stack
- `xfer` enter a given method on a given component (equivalent of `goto`; no return possible)

`return` return from the previous `call` (two 'flavours'):

- `return` return from the most recent `call`, `f-call` or `t-call`
- `throw` throw an exception

`create` create a new component instance of an installed implementation type

`destroy` destroy an existing component instance

`install` install a new implementation type

`uninstall` uninstall an existing implementation type

`lock` locks a component in the GDT cache (see Section 6.5.1)

`unlock` unlocks a component from the GDT cache (see Section 6.5.1)

`switch_stack` used to change the current stack (for example, when scheduling a new thread)

`get_stack` used to get the component reference of the currently active stack (or other in chain)

`get_self` returns the calling component's reference

`get_type` returns the reference of a given component's type

`set_type` switch the type of a component (for use with hot-swapping implementations)

`set_desc` used to manage the segment descriptor associated with a component

`fault` this method must be called if the ORB causes a hardware protection fault

`linear` used to either give linear space to, or take free linear space from, the ORB

To provide a protected system, the code-scanner must consider most ORB methods privileged; user components must be forced to direct via the library operating system to call privileged ORB methods (see Section 6.3.4 for an example). The only ORB methods that are guaranteed to be safe for any component to call are the inter-component communication methods: `call`; `f-call`; `t-call`; `xfer`; `ret`; and `throw`, as well as the read-only methods: `get_self`; `get_type`; and `get_stack`.

6.5 ORB Implementation Details

This section gives an overview of the less obvious implementation techniques used to realise the Go! ORB. An overview of the main ORB data structures is given, along with a description of the ORB's management of object references, dynamic memory allocation and hot swapping.

6.5.1 Object References

Section 5.3 describes a model in which each component instance can be identified by its data-segment selector, and each type by its code-segment selector. However, IA32 segment selectors are 16 bit, of which only 13 are used are used to identify the segment. This means that a maximum of 8,191 segments (and therefore components) are addressable at any one time[8] (selector 0 is reserved as the 'null selector'). In fact, if a component's data-segment selector is used to address it, significantly less than 8,191 components are available, since 2 segments are required for each type. This maximum number of components is likely to be insufficient for many systems, and so a mechanism must be found to extend this addressing. The solution employed is to associate a 32-bit *reference* with each segment. This reference (known as an *ObjRef*[9]) is independent of the component's data-segment selector. In effect, the GDT (Global Descriptor Table — the IA32's segment descriptor table) becomes a *cache* of up to 8,191 segment descriptors.

The ORB maintains a table known as the Component Descriptor Table (CDT), which contains *all* ObjRefs in the system. The CDT maps ObjRefs to selectors — the relevant selector if the given ObjRef is valid and cached in the GDT; an invalid selector otherwise. The entries indicating that an ObjRef is uncached or invalid are chosen so that attempting to load a segment register with a segment not cached in the GDT results in a hardware protection fault. The ORB responds to faults triggered by ObjRefs not cached in the GDT by retrying the operation after loading the relevant ObjRef into the GDT. If necessary, ObjRefs are rejected from the GDT (currently using a FIFO policy). Faults due to a component's attempted use of an *invalid* ObjRef (rather than a valid but uncached one) result in the ORB throwing an `xcp_orb_noref` exception to said component (see Section 6.3.3). Further details are presented in [66].

Figure 6.7 shows an example GDT and CDT, and an example translation for ObjRef 2. ObjRef 2 points at the third entry in the CDT. The third entry in the CDT contains selector 5, indicating

[8]The use of Local Descriptor Tables would double this, but LDTs are not relevant to SISR's 'selector as a capability' model.

[9]ObjRefs are used to identify all segments in the system (for example, types are identified by their code segment's ObjRef), hence the term ObjRef and not CompRef.

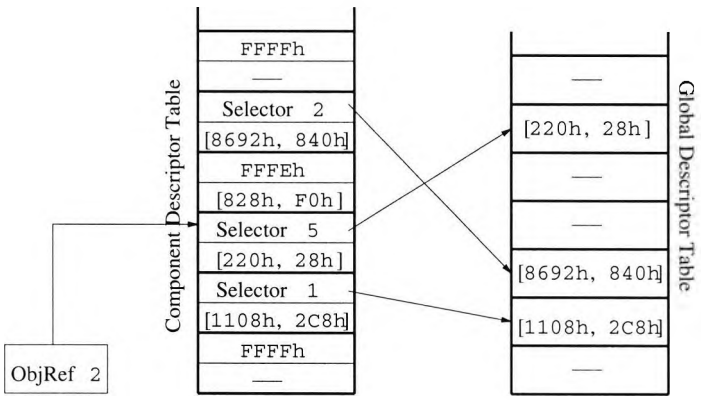


Figure 6.7: 32 bit ObjRefs with 16-bit selectors

that this ObjRef is cached in the GDT at entry 5. That is, the CDT shows that ObjRef 2 maps to selector 5. The diagram also shows that 2 other segments are cached in the GDT, with ObjRefs 1 and 4. ObjRefs 0 and 5 are invalid, since their selector fields in the CDT are FFFFh. ObjRef 3 is valid but not cached in the GDT, since its selector field in the CDT is FFFEh. Note that the component's data-segment descriptor from the GDT is duplicated in the CDT. This allows the descriptor to be reloaded into the GDT when an an ObjRef not cached in the GDT is referenced. Note also that invalid entries in the CDT shown in Figure 6.7 have their descriptor field undefined, as do unused entries in the GDT. This is because there are no circumstances under which these descriptors will be referenced.

In the above example, when the component referenced by ObjRef 3 is accessed, the ORB will attempt to load the data segment register with FFFEh (ObjRef 3's selector field in the CDT). FFFEh is an invalid selector, so this will cause a hardware-protection fault, which will be received by whatever component is responsible for interrupt management. On noticing that the fault occurred in the ORB's code-segment (selector 8) the interrupt-management component must call the ORB's `fault()` method. In response, the ORB copies the component's data-segment descriptor from the CDT to a free entry in the GDT, rejecting ObjRefs from the 'GDT cache' as necessary.

Assuming the ObjRef is cached in the GDT, using the CDT in this way means that adding a level of indirection and using 32-bit ObjRefs is actually *faster* than identifying components by their 16-bit data-segment selectors as suggested in Chapter 5. This is because identifying components by their data-segment selector requires that the selector be validated before use; this validation requires a (potentially expensive) conditional branch. Replacing this conditional branch with an indirect load and hardware-protection check is faster in the common case (that is, when the ObjRef

is both valid and cached in the GDT). See Section 7.2.5 for exact figures.

This 'trick' works because IA32 is limited to 8,191 segment *selectors*, but there is no such limit on the number of *descriptors* (up to 2^{63} distinct descriptors are available).

64-bit ObjRefs were considered, but the idea was rejected because the benefits (simpler distributed object management) were not seen to outweigh the costs (more complicated and expensive communication in the general case).

The technique of using selectors designed to trigger a hardware-protection fault rather than use a software-check with conditional branch is also used to eliminate the check for an attempted return-to-zombie (see Section 6.3.2). That is, to optimise the common case (the caller not being a zombie during an RPC return) zombies' data segments are invalidated. This means that any attempt to return to a zombie will cause a protection fault when the zombie's data segment is loaded. The ORB responds to this by throwing an `xcp_orb_zombie` exception.

Because a given segment is identified by different selectors at different times, care must be taken never to load a data-segment register directly from a selector, but always via the CDT. This means that selectors must never be cached in memory or general-purpose registers. For example, if ORB methods were to push the caller's data-segment selector onto the stack at the beginning and pop them at the end, while the segment-selector is stored on the stack the segment that selector identifies might change (due to its rejection from the GDT). This applies equally to the ORB and components such as schedulers that might wish to load segment registers (even via a TSS switch [52]). Unfortunately, the IA32 instruction-set makes it impractical to always load code-segment selectors via the CDT. This means that any scheduler component must take care to lock code segments in the GDT when they are preempted.

6.5.2 Component Descriptor Table

The Component Descriptor Table (CDT) records information on all component instances, types and initial data segments. The CDT is indexed using ObjRefs, hence instances, types and intial data segments all share a common namespace[10]. Each entry in the CDT contains more than just the selector and descriptor fields shown in Figure 6.7. In the current implementation, each CDT entry contains the following fields:

[10]Note that namespace does not refer to textual names, rather numeric identifiers.

`sel` The 16 bit selector of the associated segment if this ObjRef is valid and cached in the GDT. If the ObjRef is invalid, this field contains `FFFFh`. If the ObjRef is valid but not cached in the GDT, this field contains `FFFEh`. If the ObjRef refers to a 'zombie' component, this field contains `FFFCh`.

`ref_type` If this ObjRef refers to a component instance, this is the 16 bit selector of the component's code segment if it is cached in the GDT, or `FFFEh` if uncached. If this ObjRef is invalid, this field contains `FFFFh`. This field contains `FFFDh` if the ObjRef refers to a type, and `FFFBh` if it refers to a type's initial-data segment.

`descr` The 64 bit descriptor of the ObjRef's associated segment. When the segment is cached, this field is duplicated in the GDT.

`mtable` If this ObjRef refers to a component instance or a type, this is the 32-bit offset of the component's method table within the ORB's data segment. If the component is of type `stack`, this field is overloaded to contain the address of the stack's `stack_descriptor` (see Appendix A.5).

`mcount` A duplicate of the component's 32-bit method count (also stored in the component's method table).

`call_count` The number of times the current component has been called, but has not yet issued a `return` or `xfer` (that is, the number of threads active within the component). If this ObjRef refers to a type, this field contains the amount of zero-initialised data the type contains (that is, the size of the type's BSS).

`gdt_lock` A 32-bit integer that counts the number of request to lock the component in the GDT. A non-zero value means that the component's selector will not be eligible for rejection from the GDT. However, a non-zero value does not guarantee that the selector is present in the GDT (that is, a `gdt_lock` can be taken without the ObjRef being cached in the GDT — it just ensures that if already cached, it will not be rejected).

`type_ref` If this ObjRef refers to a component instance, this field is the pointer to the CDT entry of the instance's type. If this ObjRef refers to a type, this field is the type's initial data segment ObjRef (or zero if the type has no initial data). If this ObjRef refers to an initial data segment, the field contains `FFFFFFFFh`.

6.5.3 ORB Memory Management

Since it is desirable to allow an arbitrary number of components to be created, the CDT must be free to grow and shrink dynamically. Furthermore, in order that CDT look-ups are optimally fast (in order to keep RPC overheads low) the CDT needs to be a simple, one-dimensional array. Therefore the CDT starts at the lower addresses in memory, and grows upwards. Components should be allocated from the top of linear memory, towards lower addresses (although the details are the responsibility of the library OS) — any attempt to create an instance that will overlap with the CDT will result in an `xcp_orb_nomem` exception being thrown. The ORB also needs certain dynamic structures (see below). Since memory management is delegated to the library operating system that sits on top of the ORB, the ORB requests dynamic memory by throwing an `xcp_orb_nomem` exception (see Section 6.3.3). The library operating system's component responsible for managing linear memory must intercept this exception (see Section 6.3.2 for details on exceptions and their interception). The library operating system can then determine how much memory is required by the ORB by calling the exception's `info` method, and then allocate a suitable range by calling the ORB's `linear` method.

As mentioned above, potentially expensive conditional branches are avoided by arranging that loading an uncached or invalid ObjRef's descriptor into a segment register will trigger a hardware protection fault. If loading *any* invalid ObjRef is to fault, the ORB's data segment must implement bounds checking on the CDT. This is done by matching the size of the ORB's data segment to the size of the CDT. However, the ORB's data segment also needs access to memory outside of the CDT. For this reason, the ORB has two data segments: one covering the CDT, and one covering all linear space. This is illustrated in figure 6.8: the 'static' data segment covers the CDT and ORB static data only, whereas the 'dynamic' data segment covers all linear space[11]. Since the 'static' data segment ends at the top of the CDT, the ORB can avoid checking that an ObjRef does not refer to a location that is out of bounds, instead relying on the hardware's segment-protection mechanism. This is shown in Figure 6.8.

Note that if the ObjRef used to index the CDT is negative, an out of bounds access on the CDT might still go undetected. To avoid this, the top bit of an ObjRef is cleared before it is used to access the CDT: only the bottom 31 bits of an ObjRef are significant. In fact, because each CDT entry is 32 bytes wide, only the bottom 26 bits are significant on the current implementation. Since this number is implementation dependent, the ORB interface guarantees that at least the

[11] The ORB starts at linear address 1MB. However, the ORB 'dynamic' data segment wraps round to cover the lower 1MB too.

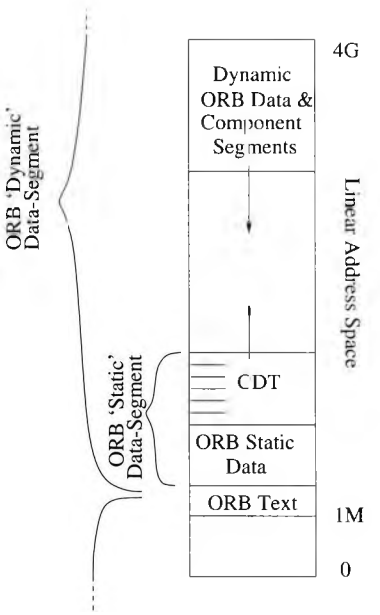


Figure 6.8: Management of linear memory

bottom 24 bits of an ObjRef will be significant, giving a maximum of approximately 16 million ObjRefs.

Inter-component RPCs require access only to the CDT and static data (except for `t-call`; see Appendix A.3.1). This means that only the 'static' data segment need be loaded during a call, saving several cycles. The exception to this is that access is required to the callee's method table, which is dynamic data. However, the method table is accessed only as part of the final jump into the callee's code segment. This requires a segment override[12] anyway since the data segment register contains the callee's data segment selector at this point (see Section 5.4.1 for details).

6.5.4 Dynamic Implementation Replacement (Hot-Swapping)

The simple architecture of Go! makes hot-swapping almost trivial. The ORB's `set_type` method simply destroys the component identified by the given reference, and instantiates a component of a new type on the same reference. It is left to the library operating system (or whatever manages hot-swapping) to ensure that the new type's interface is compatible with the old one, and manage transfer of state between implementations. This is consistent with the rest of the Go! 'philosophy': Go! provides only the infrastructure necessary to implement the more complicated primitives at higher levels.

[12] A segment override refers to use of a segment other than the default data segment.

The only slight difficulty with realising `set_type` is that threads might be executing 'inside' the outgoing component. This includes threads that have left the component through an RPC and are yet to return, and threads that are blocked. As mentioned before, the ORB is unaware of threads per se. However, the ORB maintains a `call_count` for each component that is incremented on each call and decremented on each return. This count therefore specifies how many threads are active in said component, including those threads that have left through a nested call and are yet to return. It is normally only possible to switch implementations when this `call_count` field is zero.

There are three policies if the `call_count` of the outgoing implementation is not zero when `set_type` is called:

Weak Do nothing — the hot-swapping mechanism will simply wait and try again at some later time in the hope that `call_count` will then be zero.

Strong Block all incoming calls. The component's `method_count` field is set to zero, meaning that future calls will result in an exception being thrown. The hot-swapping framework can intercept such exceptions and block the threads that triggered them. When the implementation is swapped the call that generated the exception can be retried.

Brutal Replace the implementation, regardless of the `call_count`. **This is not safe** in the general case. However, the hot-swapping mechanism or programmer may know that a swap will be safe, and so wish to force the switch anyway. Note that since `set_type` is a privileged ORB method (that is the code scanner prevents untrusted components calling `set_type`) this does not represent a protection flaw.

`set_type` therefore consumes three arguments: the reference of the component instance whose type is to be switched, the reference of the new type, and the policy (weak, strong or brutal).

6.6 Summary

This chapter has given an overview of Go!. Specifically, the programming-model, component-model, ORB interface and ORB implementation have been examined.

Go! presents programmers with a significantly different model than do conventional operating systems. Perhaps most notably, the kernel is replaced by an ORB managing components. The ORB does not offer any of the services usually associated with kernels (even very minimal systems

such as exokernels). Instead, Go! provides a component architecture and infrastructure. All services, including OS services, are implemented by components that reside within this framework.

Go! provides for protection using SISR, the novel protection model presented in Chapter 5. The ORB is minimal and lightweight, yet complete; and has shown that this new protection model can be used to implement efficient and elegant solutions. Delegating behaviour traditionally associated with operating systems to library operating systems provides the improved systems-software engineering, performance, dynamism and configurability outlined in Section 2.3. To complete the proof-of-concept, such a library operating system (known as GTE) has been constructed.

Chapter 7

Experiments

7.1 Introduction

Chapter 6 documented the development of Go! and GTE. In effect, the development of Go! and GTE was an experiment in itself (the results of which demonstrated that development an OS with protection based on SISR is practical). This chapter documents experiments more specific than this 'proof-of-concept'.

There are three types of experiments presented here: temporal performance, space requirements and stability. Temporal performance measures the time taken to perform various operations. Space requirements document how much memory various components and operations require. Stability is measured by a 'stress test', designed to load the system heavily, with the intention of exposing any bugs.

Most temporal experiments are 'micro-benchmarks' — that is, they measure specific, limited operations of the ORB. Wider benchmarks are not used for two reasons. Firstly, the *focus* of this work is about providing a lightweight protection model: the performance of a protection model itself is most accurately measured through micro-benchmarks. Secondly, since Go! is a proof-of-concept system only, it is not complete enough to run more wide-ranging benchmarks such as the time taken for a compilation or to service an HTTP request.

There are two figures for each temporal performance measurement: best case and worst case. The best-case results measure the time taken when all memory references hit the level-1 cache and branches are in the 'Branch-Target Buffer' (that is, branch-prediction succeeds). Worst-case measurements are made immediately after the cache and branch-target buffers have been flushed.

No average-case timings are given, because what constitutes an *average* case depends on Go!'s use. Assuming high cache-hit rates and branch-prediction success, the average case can be assumed to be close to the best case.

All experiments on Go! were conducted on an Intel Pentium P54C running at 90 MHz, with 32 MB of 90 ns EDO DRAM. The P54C uses 8K two-way set-associative caches for both code and data. All results are in machine cycles, measured using the Pentium's `rdtsc` instruction (an instruction intended specifically for obtaining cycle-accurate measurements for help with micro-benchmarking).

In addition, some experiments were conducted to measure the cost of the code transformations necessary to allow code containing variable-length instructions to be scanned (as outlined in Section 5.7.2). Unlike other experiments, these were not conducted on Go! since they are not measuring *operating system* performance.

7.2 ORB Performance

This section documents experiments carried out on the ORB, and their results. The temporal performance is measured for all ORB methods. For the more 'interesting' ORB methods, temporal performance is *predicted* and compared against the measurements. These 'interesting' methods are those that are core to the protection model — more specifically the times required for context switching (that is, `call/return`, `f-call/return`, `t-call/return` and `xfer`). The performance of those ORB methods needed for completeness only (for example, `get_self` and `switch_stacks`) is not so interesting in the context of this work, and so only measured results are reported in the interests of conciseness.

The performance of the RPC `call` and `return` operations is documented first. Then RPC optimisations are documented along with exception-throwing latencies, followed by the costs of a 'GDT miss'. Finally, for completeness, a full list of measured performance for all other ORB methods is presented.

7.2.1 RPC Latency

The most important performance characteristic of a protection model aimed at supporting fine-grained protection with high performance is the 'null-RPC time'. This is the time taken to call a method on a server component that takes no inputs, performs no work and produces no outputs:

only the overheads of crossing protection boundaries are measured.

As detailed in Section 6.3.1, there are several ways a client may call a method on a server. Recall that the `xfer` operation is one-way only, hence the `xfer` experiment measures only the time to `xfer` to a method on another component, not to return. The results for `call`, `t-call` and `f-call` report round-trip latencies: the time taken to invoke the callee's method *and return* to the caller.

This section gives the theoretical performance of null-RPC followed by the measured performance. The theoretical performance is calculated by decomposing the operation into its constituent parts, and using the Intel Pentium Optimisation Manual [53] to obtain the documented cycle counts for each operation. Note that for simplicity's sake, processor implementation complexities such as dual dispatch, data dependencies and pipeline stalls are not taken into account (hence the theoretical performance is *not* a theoretical maximum performance).

The theoretical and measured results are compared and reasons are suggested for any significant discrepancies.

Theoretical Performance

As outlined in Section 5.4.1 there are 11 steps to issuing a call and 7 steps to return (although the algorithm here changes *slightly*, due to the CDT and ObjRefs — see Section 6.5.1). Table 7.1 gives estimated cycle counts for each step. Similarly, Table 7.2 shows the theoretical performance

Operation	Cycles
load ORB data-segment selector into the data-segment register	3
push caller's reference onto the stack	2
increment callee's call-count	3
validate callee method number is within callee's method count	2
push details of previous call onto the stack (e.g. previous stack size)	5
shrink stack by manipulating segment-descriptor tables	12
increment call-depth associated with current stack segment	3
look-up target code segment and offset in method table	2
load callee data segment into the data-segment register via the CDT	4
place caller ObjRef in general-purpose register (authenticate)	1
jump into callee code segment at offset indicated by method table	3
total	40

Table 7.1: Theoretical performance of the `call` primitive.

of the `return` primitive when used to return from a `call`. This gives a total of 77 cycles for a `call/return` pair. To predict the complete null-RPC time one must simply add this figure to the time taken for the caller and callee to enter the ORB (that is, the time taken for the inter-segment

Operation	Cycles
load ORB data-segment selector into the data-segment register	3
determine type of return (assuming return from `call`)	4
re-grow stack by manipulating segment-descriptor tables	12
pop details of previous call from stack (e.g. previous stack size)	5
decrement call-depth associated with current stack segment	3
decrement callee's call-count	3
pop caller's ObjRef from the stack and load data segment via the CDT	4
issue an inter-segment return, returning control to caller	3
total	37

Table 7.2: Theoretical performance of a `return` from a `call`.

jumps to the ORB's `call` and `return` routines). Intel documents inter-segment jumps at taking 3 cycles each on a Pentium, giving a predicted null-RPC overhead of 83 cycles.

The measured latency for best-case null-RPC round-trip is somewhat higher than the predicted performance, at 252 cycles. This is considerably worse than predicted, mainly due to the two stalls caused by writing to the GDT (see Section 6.3.1).

7.2.2 RPC Variations and Optimisations

As mentioned in Section 6.3.1, Go! provides several variations on the standard RPC `call` primitive: `f-call`, `t-call` and `xfer`. This section presents the predicted and actual temporal performance of these primitives.

Predicted `f-call` Latency

The algorithm and associated cycles for `f-call` are given in Table 7.3. Table 7.4 gives the the-

Operation	Cycles
load ORB data-segment selector into data-segment register	3
push caller's ObjRef onto the stack	2
increment callee's call-count	3
validate method number is within callee's method count	2
push details of previous call onto the stack (e.g. previous call-depth)	5
find the 'next' stack in chain	3
switch to the next stack in chain	6
place caller's ObjRef into a general-purpose register (authenticate)	1
look-up callee's method table	2
load callee's data segment into data segment register via the CDT	4
jump to callee's entry point	3
total	34

Table 7.3: Theoretical performance of the `f-call` primitive.

oretical performance of a `return` from an `f-call`. This gives a total of 71 cycles for null-RPC

Operation	Cycles
switch to ORB's data segment	3
determine type of return (assume return from `f-call`)	2
find 'previous' stack in chain	4
switch to 'previous' stack in chain	7
pop details of previous call from stack (e.g. previous stack size)	5
decrement callee's call-count	3
pop callee's ObjRef from the stack, and load data segment via the CDT	4
issue inter-segment return to caller	3
total	31

Table 7.4: Theoretical performance of a `return` from an `f-call`.

using `f-call` (including the 6 cycles required to enter the ORB at its `f-call` and `return` entry-points). `f-call`'s measured latency is 73 cycles. These slight differences can be attributed to the subtleties of running the tests on real hardware, such as dual dispatch and inter-instruction data dependencies).

`t-call` Predicted Latency

The t-call operation is similar to `call`, except that the callee's stack is not protected. However, some protection is still necessary in order to prevent a component called with `t-call` returning to an arbitrary location in another component. Table 7.5 gives the theoretical performance of the `t-call` operation. Table 7.6 gives the theoretical performance for a `return` from a `t-call`.

Operation	Cycles
load ORB data-segment selector into data-segment register	3
push caller's ObjRef onto the stack	2
increment callee component's call-count	3
validate caller ObjRef and method number	4
obtain element for 't-call list' to record return information	15
record return information and link element into the stack's t-call list	10
increment call-depth associated with current stack segment	3
look-up callee's method table	2
place caller ObjRef in general-purpose register (authenticate)	1
load callee's data segment into data-segment register via the CDT	4
jump to callee's entry point	3
total	50

Table 7.5: Theoretical performance of the `t-call` primitive

Including the 6 cycles required to enter the ORB by the caller and callee, this gives a total of 101 cycles: not signficantly different from the measured performance of 96 cycles.

Operation	Cycles
load ORB's static-data-segment selector into the data-segment register	3
determine type of return (assume return from `t-call`)	4
restore the return information stored in element from 't-call list'	10
free the return element to ORB's internal memory	15
decrement callee's call-count	3
decrement the call-depth associated with the current stack	3
pop callee's ObjRef from the stack, and load data segment via the CDT	4
issue inter-segment return to callee	3
total	45

Table 7.6: Theoretical performance of a `return` from a `t-call`.

`xfer` Predicted Latency

The `xfer` primitive has relatively little work to perform since no stack protection or recording of return information is necessary. The algorithm and associated costs are shown in Table 7.7.

Operation	Cycles
load ORB's static-data-segment selector into the data-segment register	3
validate method number within callee's method count	2
decrement caller component's call-count	3
increment callee component's call-count	3
look-up callee's method table	2
place caller's ObjRef in general-purpose register (authenticate)	1
load callee's data segment into data-segment register via the CDT	4
jump to callee's entry point	3
total	21

Table 7.7: Theoretical performance of the `xfer` primitive.

This is a total of 21 cycles, plus 3 cycles required to enter the ORB (once), giving a predicted latency of 24 cycles. Again, not significantly different than the measured latency of 28 cycles.

7.2.3 Control-Transfer Measurements

To summarise, the various control-transfer primitives' latencies are presented in Table 7.8.

Primitive	*Worst Case (Cycles)*	*Best Case (Cycles)*
`call/return`	1859	252
`t-call/return`	1693	96
`f-call/return`	1091	73
`xfer`	738	28

Table 7.8: Summary of null-RPC times using the various primitives offered by Go!

Note that despite not providing protection, `t-call` is still slower than `f-call`. This is because

`t-call` must ensure that control returns to the appropriate component, while not protecting the stack (which means that the ORB must store the return address internally).

7.2.4 Exception Latency

As detailed in Section 6.3.2, Go! supports software exceptions. Throwing an exception is much like issuing an RPC return, except that control does not return normally, but via interrupt vector 30h.

The results for `throw` measure the time taken between throwing the exception, and receiving the interrupt. However, since this involves unwinding the stack, the time will depend on what type of call was last issued (`call`, `f-call` or `t-call`). For this reason there are three pairs of results for `throw`: one for each type of call. Table 7.9 shows the time overheads for throwing an exception predicated on whether a `call`, `f-call` or `t-call` was most recently used.

Last Call	*Worst Case (Cycles)*	*Best Case (Cycles)*
`call`	465	116
`f-call`	318	63
`t-call`	409	86

Table 7.9: Measured exception latencies depending on the most-recent call primitive.

7.2.5 GDT miss

It is reasonable to assume a high hit-ratio of components cached in an 8K slot GDT (although exact figures will depend on the application). Still, it is important to ascertain the penalty for a 'cache miss' (that is, the penalty involved when invoking a method on a component with its data-segment descriptor not cached in the GDT). It is even more important that the scheme does not adversely affect the inter-component method call latency when a component's data-segment descriptor is cached in the GDT (that is, the common case).

The experiments examine the time taken to transfer control between two components using Go!'s `xfer` primitive [68]. Three experiments were conducted: one for an implementation using data-segment selectors to identify components (as suggested in Chapter 5), and one each using 26 bit ObjRefs (as described in Chapter 6), where the callee-component's data-segment descriptor was and was not cached in the GDT[1]. Table 7.10 presents the results. The experiments show

[1] the caller component will always be in the GDT since its data-segment selector is loaded into the data-segment register.

Primitive	*Time (Cycles)*
Using 16 bit selector	30
Using ObjRef cached in GDT	28
Using ObjRef not cached in GDT	1080

Table 7.10: The overheads of extending the GDT using the CDT.

a hit of around 30 fold when a call is made on a component with its data segment not cached in the GDT. Given that taking a miss on the GDT is analogous to taking a page-miss on a virtual-memory system, the miss penalty is quite acceptable (as with virtual memory systems, components with critical or sensitive performance can be locked in the GDT). More importantly, indirecting ObjRefs via the CDT does not affect performance in the common case (other than a slight improvement!).

7.2.6 Other ORB Methods

This section presents the times taken for all other ORB methods. Where the time is dependent on the parameters, the time is taken for the least amount of work possible. For example, `create` implicitly calls a component's constructor: times are given where the constructor performs no work other than issuing an RPC return. This way the results measure the impacts of Go!'s protection model, rather than (say) the performance of copying large amounts of data. Table 7.11 gives a complete list of the measured overhead of each ORB method.

7.2.7 Comparison with other OSs

To set the results of the experiments outlined above in context, they are compared to similar experiments conducted on other operating systems. Unfortunately, Go!'s novel architecture means that most ORB methods do not have direct equivalents in other operating systems. However, most systems do support intra-machine RPC and one-way cross-protection-domain communication, enabling comparison of Go!'s `call` and `xfer`. Also, Go!'s `get_self` operation is analogous to UNIX's `get_pid`.

Go!'s performance was compared with the following OSs': *Linux* (an example of a widely used, commodity OS), *Mach* (an example of a first-generation μ-kernel), *Pebble* (an example state-of-the-art, component-based OS), and *L4* and the *Xok* exokernel (two other example state-of-the-art research systems). While this list is obviously not exhaustive, it includes at least one example of all the main categories (it also includes the three fastest kernels the author is aware of).

Operation	*Worst Case (Cycles)*	*Best Case (Cycles)*
`create`	2795	673
`create` (`null` component)	1247	131
`create` (unlinked stack)	2065	320
`create` (linked stack)	2657	416
`destroy`	2043	522
`destroy` (`null` component)	1509	289
`destroy` (unlinked stack)	1811	444
`destroy` (linked stack)	2110	497
`install`	2298	965
`uninstall`	564	187
`get_self`	435	17
`get_type`	524	22
`get_stack`	650	22
`switch_stacks`	641	34
`lock`	560	29
`unlock`	650	28
`sel2ref`	432	23
`reject`	337	23

Table 7.11: Measured cycles-counts for every ORB method.

Figures have been taken from various publications [86, 29, 32, 38, 74], as well as experiments. When possible, the same hardware is used (that is, where available, results are taken for the Pentium). However, such figures are unavailable for Mach and Pebble. Figures for OSs running on different platforms are still useful, since they provide a comparison for 'orders of magnitude'. The only non-Intel OS that gets within an order of magnitude of Go! is Pebble. However, it is likely that Pebble would take significantly longer on the Intel, since for a one-way IPC L4 takes just 86 cycles on the MIPS R4600, but 121 cycles on the Pentium (Pebble takes 114 cycles on the MIPS R5000).

The best-case of Go!'s figures are taken, since the figures taken for other systems are best-case too. Null-RPC corresponds to an `f-call/return` pair for Go!. 'one-way xfer' corresponds to the lowest time for cross-domain communication in each system (`xfer` for Go!). Table 7.12 presents the timings obtained for Go! and other OSs.

OS	*Platform*	*Null RPC (Cycles)*	*xfer (Cycles)*	*get_pid (Cycles)*
Linux	Pentium	42,750	3,990	225
Xok	Pentium	4,440	N/A	176
Mach 2.5	MIPS	3,000	1,600	276
L4	Pentium	288	121	N/A
Pebble	MIPS	N/A	114	N/A
Go!	Pentium	73	28	17

Table 7.12: Temporal overheads of Go! with comparison to other OSs.

Table 7.12 shows that a null-RPC on Go! out-performs commodity systems by around almost 3 orders of magnitude, and the fastest, state-of-the-art research systems around four-fold.

7.2.8 Spatial Performance

The ORB uses very little memory. Perhaps most impressive is the space required per component: just 32 bytes overhead (the CDT entry). This is around *two orders of magnitude* improvement over traditional, page-based protection models.

The ORB itself is also small. The code section is just 12K, with 2K of static data. The size of the ORB's data segment depends mostly on the size of the GDT (configurable at compile time from 1K to 64K). The larger the GDT, the fewer selector misses will be incurred — this is a conventional trade-off of space against time.

7.3 GTE Performance

Unlike the Go! ORB, GTE has not been aggressively optimised. Given the number of GTE operations, and that most have not been optimised, an extensive break-down of the timing for each GTE method would be laborious and of little interest. Instead, this section presents experiments that look at performance in a wider context, such as scheduling or code scanning overhead. The size of each GTE component is also given, to give an idea as to the granularity with which GTE is decomposed.

7.3.1 Component Sizes

As mentioned, GTE consists of 10 distinct components. This section lists the code size, data size, and BSS size for each component. Code size corresponds the number of bytes a component's code segment consumes, data size is the amount of statically-initialised data, and BSS the amount of zero-initialised data. Table 7.13 presents the size of each component.

7.3.2 `sched` overhead

Each timer interrupt schedules a new thread, resulting in the interaction of 7 distinct components (the outgoing and incoming thread and stack components, the interrupt-dispatcher, the scheduler and the ORB). Contrasting this with conventional systems (including μ-kernels) that incorporate

Component	*Code (bytes)*	*Data (bytes)*	*BSS (bytes)*
comp_lib	5,664	1,742	0
mem_mgr	2,432	704	2^{32}
idisp	6,912	944	832
sched	3,040	512	196,656
scan	4,288	3,488	1,067,616
xcp_mgr	560	240	0
thread	176	128	16
video	1,344	80	0
keyb	1,056	288	80
cli	13,974	800	3,200

Table 7.13: The Size of Each Component Type

these 7 distinct components into a single entity (namely, the kernel), it is clear that this approach might incur performance problems due to the increased context switching.

An experiment was conducted to assess whether the decomposition of scheduling introduces unacceptable overhead. The experiment was conducted with two runnable threads, operating with interrupts disabled. A timer interrupt was simulated by issuing an `int 32` instruction from the first thread, and the time was measured for the second thread to receive control. Using the Pentium's `rdtsc` instruction to count processor cycles, this time was found to be just under 2,500 cycles.

With a timer-interrupt frequency of 100 ticks per second, this is an overhead of less than 0.3% on the 90MHz test machine used. Furthermore, relatively little of this overhead can be attributed to the decomposition. Firstly, the single interrupt causes two threads to be scheduled (the scheduler-thread, as well as the second runnable thread), it would be trivial to reduce this to just one. Secondly, for simplicity the current implementation uses the Intel's TSS hardware-task-switching mechanism, known to perform poorly (most commodity OSs save and restore context purely in software). Thirdly, the scheduling code generally is not heavily optimised. Implementing these three optimisations is likely to reduce significantly this already small overhead.

7.3.3 Code Scanner

As mentioned in Section 5.7.2, code scanning on architectures with variable-length instruction sets is non trivial. Table 7.14 shows the time taken to scan all components within GTE. Table 7.14 shows that as predicted, the code-scanner has linear complexity, at approximately 90 cycles per byte of code scanned. The *cycles per byte* measurement is less variable than the cycles per instruction count, since scan runs are issued from 8-byte offsets, regardless of how many

Component	*Bytes of Text*	*Instructions*	*Cycles*	*per Byte*	*per Instruction*
`thread`	176	56	16,088	91	287
`stress`	192	53	15,834	82	298
`xcp_mgr`	784	252	65,709	83	260
`keyb`	1056	366	97,511	92	266
`video`	1,344	422	118,855	88	281
`mem_mgr`	2,432	753	255,465	92	339
`sched`	3,040	901	274,989	90	305
`idisp`	6,921	2,374	633,330	91	266
`cli`	13,794	3,816	1,185,377	86	310

Table 7.14: Scan Times of the Various GTE Component Types.

instructions are actually present. That is, the same piece of binary code will contain different numbers of x86 instructions depending on at what offset code is considered to start.

7.4 Stability

To measure the overall system's stability, a stress-test was conducted. This test consisted of creating 1,000 instances of a `stress` component, and 1,000 threads. Each `stress` component either called into one of the others using `call`, `f-call`, `t-call` or `xfer`, or returned (the choices are made randomly). When stack space was low a return was forced to prevent stack overflow.

Note that 'random' in this context obviously means pseudo-random: the lower bits of the sum of the cycles elapsed since reboot are used. This is known to be a poor general-purpose pseudo-random number generation algorithm, but is "good enough" in this context. Furthermore, to produce genuinely random (and asynchronous) stimuli, key-presses were generated and processed at various times during the test.

This stress test was run for 4 days without failing (the choice of 4 days is somewhat arbitrary, but certainly long enough to give some confidence in the stability of a prototype system). Note that the test did not fail after 4 days, but was terminated.

The stress test is important because it shows that performance figures can be taken seriously — bug fixing of the critical paths would likely adversely affect performance.

7.5 Dynamism

An experiment was conducted to demonstrate Go!'s suitability for dynamic implementation replacement. That is, replacing a component's implementation for a new one with minimal inter-

ruption of said component's service.

As stated in Section 2.3.4, supporting the transfer of state during the evolution of a component is *not* a goal of this work. However, the component framework must support the hot-swapping of implementations, and allow the programmer or layers above to handle the transfer of state. Therefore, it must be possible to 'hot-swap' components on top of Go! As a demonstration of this, GTE's `keyb` component (which has UK layout) was dynamically replaced by a version for US layout keyboards.

7.6 Code Scanning and Variable-Length Instructions

As mentioned in Section 5.7.2, code scanning is non-trivial for variable-length instruction sets, but the problem can be overcome by inserting a few instructions into the code at compile time. This allows the code scanner more easily to verify that code containing variable-length instructions will not execute privileged instructions, even in the light of indirect jumps.

Obviously, such insertion of instructions will have adverse impacts on code size and speed. The final cost of this will depend on the program being modified — a program with a large number of indirect jumps will be more affected. To obtain the approximate costs, the *Dhrystone* benchmark [111] was performed with and without application of the modifications described. Dhrystone was chosen because it is a well-known yet relatively simple test that models typical code use, incorporating function calls and switch statements. The benchmark was run on an Intel Pentium III with 128MB of SDRAM running Linux[2].

Table 7.15 gives the "Dhrysontes per Second" for the modified and unmodified Dhrystone benchmark — that is, the temporal overhead of the modifications is measured for typical code.

GCC Optimisation	Modified Code	Unmodified Code	Slowdown
Off	616,649	636,942	3.2 %
On (-02)	871,912	920245	5.3 %

Table 7.15: Temporal overheads of modifications for CISC code-scanning.

Table 7.16 shows the spacial overheads incurred by the Dhrystone benchmark when it is modified for use with code-scanning. Sizes are in bytes.

The results in Tables 7.15 and 7.16 show that code modified so as to enable code-scanning on

[2]Note that it was not necessary that this experiment was run on Go! (or any other particular system) since measurements were obtained using a single process only — that is, the operating system code was not being benchmarked; only "application code". The Linux platform was chosen for convenience only.

GCC Optimisation	Modified Code	Unmodified Code	Increase
Off	9,908	9,492	4.4 %
On (-02)	8,840	8,488	4.1 %

Table 7.16: Spacial overheads of modifications for CISC code-scanning.

architectures with variable-length instructions incurs low overheads.

Even so, atypical code may incur worse penalties when modified for use with code-scanning. In such cases, it will still be possible to run those particular code sections with the processor in user mode. That is, if the costs of code-scanning on variable-length instruction-set architectures are seen to outweigh the benefits, code-scanning need not be used for such components.

7.7 Summary

This chapter has presented several experiments and their results. Performance experiments have provided very pleasing results: Go! is approaching 3 orders of magnitude faster than commodity systems, and one order of magnitude faster than even the leanest research systems. The speed improvements compared with lean research systems such as L4 are due to the new protection model rather than small optimisations and tweaks — this can be asserted since L4 is already a heavily optimised system.

A stress-test has also shown that the system is stable. Without this stability, the impressive performance results would be meaningless, since bug-fixes would likely adversely affect performance.

Chapter 8

Conclusion

8.1 Introduction

This thesis has presented a novel operating system protection model, and then presented a proof-of-concept implementation. With this new protection model (known as SISR), user-level code is prevented from executing privileged instructions not through a special processor mode, but by scanning the code before it is loaded to ensure no privileged instructions are present. Privileged instructions are those that can be used to circumvent protection.

This chapter draws conclusions from this work, with reference to the goals set out in Section 2.3. Each goal is shown to have been met, thus the work presented here can be considered a success. Possible directions for future work are also given.

8.2 Thesis Review

The first and most obvious conclusion to draw is that the "proof-of-concept implementation proved the concept". That is, the implementation of Go! and GTE (see Chapter 6) have shown that the development of a system using SISR is feasible. However, Go! and GTE have shown more than this — they have shown that the new model is *useful*. Here, 'usefulness' is measured in terms of meeting the goals outlined in Chapter 2, specifically:

8.2.1 Software Engineering

Most operating systems treat the kernel as an inseparable, immutable whole. By decomposing the kernel, Go! has simplified significantly the task of systems-software development. Traditionally, kernels have provided many different services, including (at least): protection-domain management and cross-domain communication; interrupt dispatching; memory management; fault-management; paging; (preemptive) scheduling; device management and security policy. Some recent kernels such as μ-kernels and exokernels provide these services only at a rudimentary level, requiring user-level 'servers' to perform most of the work. However, the kernel still needs to provide some sort of support. For example, although many μ-kernels allow user-level applications to define their own page-replacement policies, the kernel still takes page-faults and 'farms them out' to applications.

Go! is different. Its component-model meets all four of the key requirements identified in Section 1.3 in that it is orthogonal, consistent, pervasive and lightweight. As shown with GTE, different services can be truly separated into distinct components so that one component provides just one service. For example, different components offer one of: protection-domain management and cross-domain communication (the ORB); interrupt-dispatching (`idisp`); memory-management (`mem_mgr`); fault-management (`xcp_mgr`); preemptive scheduling (`sched` and `thread`); device management (`video` and `keyb`). Paging and explicit security management have not been added since they are beyond the scope of this project, but it is easy to imagine how `pager` and `security` components could be developed.

It has been shown that implementing fault containment modules at the same boundaries as the conceptual abstractions of a design improves reliability of software and improves debugging [65]: the low overheads of SISR permit this while maintaining performance.

In some respects Go! is similar to the exokernel approach, except that the concept of a minimal kernel is taken to its logical conclusion — in effect a 'zero-kernel' has been constructed. However, there are major differences between Go! and exokernels, most notably that an exokernel provides a base on top of which programs and library operating systems can be multiplexed in a protected fashion. Go! provides just a component architecture — components such as a code-scanner and interrupt dispatcher collaborate to provide protection. Note that there is no impediment to developing one or more components that sit on top of Go! to provide an exokernel.

8.2.2 Performance

The experiments presented in chapter 7 have shown that Go! out-performs even the leanest research operating systems significantly. This lightweight protection can be used to decompose systems to a finer granularity while maintaining end-to-end performance. Section 2.3.2 specified that the new system allow round-trip, intra-machine, null-RPC in less than 150 cycles. Go!'s `f-call` provides this in just 73 cycles — less than half the maximum latency specified. Similarly, Section 2.3.2 specified that placing a component in its own protection-domain must incur no more than 100 bytes overhead — Go! imposes just 32 bytes overhead per protection domain.

This performance also improves software engineering and reliability through the fine granularity made possible. While traditional protection models do not place architectural restrictions on the decomposition of user-level services, performance considerations render fine-grained decomposition impractical on conventional systems.

8.2.3 Configurability

Decomposition of the TCB as achieved with GTE means that the operating system can be highly configurable. During development, implementations of GTE components have been radically altered, with no changes required to other components.

8.2.4 Dynamism

The Go! ORB has support for dynamic replacement of implementations while such components' clients remain completely unaware (providing the interface contract is still realised). The experiment in Section 7.5 demonstrates the ORB's suitability for such behaviour.

Note however that transfer of state between implementations is left to the programmer or library operating system to manage — Go! provides only the *infrastructure* for hot swapping.

8.2.5 Language Independence

The code-scanning model introduced in Chapter 5 is concerned only with machine code and so is language-independent. Despite being implemented solely in C, C++ and x86 assembler, Go! presents no impediment to implementing components in any compiled language (although the best strategy for implementing interpreted languages on Go! is less clear — see Section 8.3.3).

8.2.6 Software Reliability

The finer granularity of protection will also allow systems to be more reliable through fault containment. There are three main types of software behaviour: crashes, erroneous results and correct operation. Erroneous results are by far the most dangerous of the three since they are difficult to detect and can propagate throughout the program. Realising protection at a finer granularity will make it more likely that programming errors are caught, and so move instances of 'erroneous results' into the 'crashed' category, yielding safer software. This is corroborated in [99] that shows how most erroneous results (particularly in the operating system) are the result of undetected erroneous memory references. The finer granularity of protection made possible by SISR will mean that more erroneous memory references will be detected.

8.3 Future Research

The development of Go! and GTE has brought to light many further research directions that are beyond the scope of the work presented here. These are documented in this section, along with hints towards possible solutions.

8.3.1 Distribution

The Go! ORB has no notion of distribution. However, it is designed in such a way that a number of distribution mechanisms and policies may be constructed on top. The simplest strategy for implementing distribution on top of Go! is to use *proxies*. However, the proxy method suffers from a number of drawbacks, mainly due to the requirement of extra component instances (proxies can also introduce 'existential issues'; should an equivalence test between a reference to a server component and a reference to one of its proxies produce true or false?).

One solution is to intercept exceptions thrown by the ORB in response to the use of an invalid component reference. A distribution-management component can be instantiated on each machine, which intercepts all such exceptions. In response to such an exception, the distribution-management component must determine whether the reference is genuinely invalid, or if the component with the faulting reference exists on another machine. In the case of the latter, the distribution manager must forward the call over the network to the relevant remote machine.

8.3.2 Resource Leaks

Decomposition of a system into hundreds of thousands of components is likely to create administrative problems — 'leaky' systems can quickly clog up resources. Traditional garbage collection [1] is not possible with SISR since it is impossible to determine which words in memory contain component references.

One solution is to introduce a concept of ownership, such that a component *owns* any components it creates. This will lead to a tree of component ownership: an administrator could manually 'prune' this tree to free up resources, particularly if destruction of an instance causes the destruction of all components *owned* by that component. Furthermore, the number of components a component is permitted to own (and thus create, directly or indirectly) can be strictly limited. The notion of ownership would most likely be built into the library OS component responsible for linear-memory management.

8.3.3 Interpreted Languages

The suitability of SISR's component-based architecture with regards to interpreted languages is not clear. The interpreter will need direct access to its components' code and data, possibly requiring many segment-register overrides, and thus adversely effecting performance. Furthermore, the benefits of protecting interpreted components from one another are not clear.

One solution is to have one large component contain the interpreter and all its programs' objects. For example, a `java` component might contain the code of the interpreter, and the state of all Java objects in a running Java program. In this scheme there would be one type per interpreted language, and one instance per interpreted program. The interpretor might create proxy components for communication between interpreted programs' objects and the 'outside world'.

Just In Time compilation raises more issues. Depending on the exact nature of the compilation, component types could be created dynamically (such types would need to be passed through the code scanner before being installed). In the case where self-modifying code is genuinely required, such components will need to execute in user mode (or the JIT compiler be trusted[1]).

[1] Proof-carrying code [80] promises to help here.

8.3.4 A Developers' ORB

Because the Go! ORB does not handle interrupts or exceptions, programming the early stages of library operating systems can be tedious. This is because until a library OS capable of receiving and handling faults is up and running, any faults generated cause a reboot (this includes attempted use of invalid ObjRefs; see Section 6.5.1). Since one of Go!'s goals was to improve systems-software engineering, this is less than ideal. To remedy this, a developers' ORB could be constructed that handles exceptions internally, unless overridden by the library operating system. This way, faults generated by errors in the core of the library OS can be caught and their details reported to the library OS programmer — a vast improvement over a machine reset.

8.3.5 ORB-Level Security

The Go! ORB implements authentication only (by supplying the caller's reference on each call); any authorisation mechanisms must be built outside (see Section 5.4.1). This has the advantage of decomposing security from protection, but the disadvantage that the security framework will likely slow down RPC, due to indirections via some 'authoriser' component. This performance issue could be overcome by adding a 'secure-call' (`s-call`) ORB method that will provide authorised RPCs.

This could be implemented by adding the notion of secure interfaces. These are ObjRefs onto a component with method tables which have a subset of methods available. For example, there might be two secure interfaces onto a file component, a read/write one with all methods available, and read-only one where the `write` method-table entry points not to the real `write` entry point, but to an alternative method that simply throws an exception.

Secure interfaces do not have their ObjRefs stored in the CDT as normal and so cannot be called using `call`, `f-call`, `t-call` or `xfer` — the `s-call` primitive must be used. Each component capable of using `s-call` has associated with it an array of pointers to secure-method tables. This array is filled by calling another (privileged) ORB method, `link_secure( objref client, objref secure_inf, uint method_no )`. Establishing a pointer to a secure method table from this array is effectively establishing an authorised channel or capability to the server.

The implementation suggested should provide secure RPCs with at most only a few extra cycles overhead compared to the inter-component primitives in the current implementation of Go! Note that this is a performance optimisation only — the same architecture could be setup on the current implementation where secure calls are indirected to some security component that then

calls the real server on the client's behalf via an `xfer`. Here, the real server would only accept calls that originate from the security component.

8.4 Wider Implications

This section examines the wider implications of SISR. If widely adopted, SISR and component-based development are likely to have a significant effect on the way software is developed.

8.4.1 Implications for Software Development

Computer programs are constrained and influenced by the environment in which they run. For example, programs are developed very differently depending on whether they are written to run on massively parallel machines, vector computers, scalar machines, or some other architecture. This applies equally to the operating system as it does to the hardware architecture. If SISR is widely adopted, it is likely to affect the way that programs are developed, and even lead to the developments of entirely new types of computer program.

For example, 'A Life' software [34] works radically differently than conventional software. Millions of small components 'evolve' and interact to produce systems far more complex than humans can design. The protection model presented here could allow new types of A-life components which are truly independent and can interact in controlled ways. Similarly, SISR might have implications for agent-based software. Here, agents can be encapsulated into components, and systems interacting with an agent need not necessarily trust that agent.

Less 'grand' but perhaps more likely is that component-based systems developed using this new model will be developed differently. The low overheads will mean fine-grain decomposition is practical, and that encapsulation will be enforced. This means that software engineers will not be tempted to 'cut corners' and break encapsulation in order to effect quick fixes.

8.4.2 Implications for Chip Designs

Although not totally dependent on segmented architectures, code scanning is much more effective with segmentation. This work has shown that contrary to contemporary belief, segmentation can be a useful and powerful tool with respect to memory protection. Like most of computer science (and indeed all engineering), different strategies come in and out of vogue over time. It is not anticipated that the work presented in thesis alone will alter future microprocessors, but

the research has demonstrated a hitherto unrealised benefit of segmentation. With other similar work and the advent generally of component-based operating systems it is possible that future microprocessors' MMUs will once again offer segmentation.

8.5 Summary

This chapter has presented the conclusions of the work that lead to SISR, thoughts on future work, and possible wider implications. In so doing, the goals outlined in Chapter 2 have been shown to have been met, and SISR has been shown to offer a useful protection mechanism, particularly in the context of component-based operating systems.

Bibliography

[1] Guy T. Almes. *Garbage Collection in an Object-Oriented System.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, June 1980.

[2] Apple Computer, Inc. *Inside Mac OS X: System Overview.* Fatbrain, 2001. ISBN: 1-40052-480-6.

[3] Apple Computer, Inc., IBM Corporation, and Motorola, Inc. *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture.* Morgan Kaufmann Publishers, 1995. ISBN: 1-55860-394-8.

[4] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1987.

[5] Godmar Back, Patrick Tullmann, Wilson C. Hsieh, and Jay Lepreau. Java Operating Systems: Design and Implementation. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, August 1998.

[6] P. Barham. *Devices in a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, July 1996.

[7] J. M. Bernabeu-Auban, Phillip W. Hutto, Yousef M. Khalidi, Mustaque Ahamad, William F. Appelbe, Partas Dasgupta, and Richard J. LeBlanc. The Architecture of Ra: a Kernel for Clouds. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, volume II, pages 936–945, Kailua-Kona, HI, USA, January 1989. IEEE Comput. Soc. Press. Also Georgia Tech Tech Report [GIT-ICS-88/25].

[8] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. SPIN - An Extensible Microkernel for Application-specific Operating System Services. In *Proc. 1994 European Special Interest Group in Operating Systems (SIGOPS) Workshop*, 1994.

[9] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, volume 23(5), pages 102–13, December 1989.

[10] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson. TENEX, A Paged Time-Sharing System for the PDP-10. *Communications of the ACM*, 15(3):135–143, March 1972.

[11] Shahid H. Bokhari. The Linux Operating System: an Introduction. Technical Report TR-95-49, Institute for Computer Applications in Science and Engineering, June 1995.

[12] Grady Booch. *Object-Oriented Design with Application.* Benjamin/Cummings, 1991. ISBN: 0-8053-0091-0.

[13] R. Campbell, G. Johnston, and V. Russo. CHOICES (Class Hierachical Open Interface for Custom Embedded Systems). *Operating Systems Review 21, pp. 9-17*, pages 9–17, July 1987.

[14] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. Opal: a Single Address Space System for 64-bit Architecture Address Space. In *Third Workshop on Workstation Operating Systems, April 23–24, 1992, Key Biscayne, Florida: proceedings*, pages 80–85. IEEE Computer Society Press, 1992. ISBN: 0-8186-2555-4 (paper), 0-8186-2556-2 (microfiche).

[15] Jeffrey S. Chase, Valerie Issarny, and Henry M. Levy. Distribution in a Single Address Space Operating System. *ACM Operating Systems Review*, 27(2):61–65, April 1993.

[16] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight Shared Objects in a 64-Bit Operating System. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 397–413, 1992.

[17] David R. Cheriton. V Kernel: a Software Base for Distributed Systems. *IEEE Software*, 1(2):19–38, 40–42, April 1984.

[18] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–193, 1994.

[19] Morten M. Christensen. Methods for Handling Exceptions in Object-oriented Programming Languages. Master's thesis, Odense University, Odense, Denmark, 1995.

[20] Robert P. Colwell, Edward F. Gehringer, and E. Douglas Jensen. Performance Effects of Architectural Complexity in the Intel 432. *ACM Transactions on Computer Systems*, 6(3):296–339, August 1988.

[21] M. Condict, D. Bolinger, E. McManus, and D. Mitchell. Microkernel Modularity with Integrated Kernel Performance. Technical report, OSF Research Institute, Cambridge, MA, April 1994.

[22] Digital Equipment Corporation. *Alpha Architecture Handbook.* Digital Press, 1992.

[23] Timothy Cramer, Richard Friedman, Terrence Miller, David Seherger, Robert Wilson, and Mario Wolczko. Compiling Java Just in Time: Using runtime compilation to improve Java program performance. *IEEE Micro*, 17(3):36–??, May/June 1997.

[24] David Cutler. NT. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures: 27–28 April, 1992, Seattle, WA, USA*. USENIX, 1992. ISBN: 1-880446-42-1.

[25] P. Dasgupta, R. LeBlanc, M. Ahamad, and U. Ramachandran. The Clouds Distributed Operating System. *IEEE Computer 24*, November 1992.

[26] Drew Dean, Edward. W. Felton, and Dan S. Wallach. Java security: From hotjava to netscape and beyond. *Proceedings of 1996 IEEE Symposium on Security and Privacy (Oakland, California)*, May 1996.

[27] A. Dearle and D. Hulse. The Charm Operating System Web Pages, 1999. http://os.dcs.st-and.ac.uk/Charm/.

[28] E. W. Dijkstra. Goto Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, May 1968.

[29] R. Draves, B. Bershad, R. Rashid, and R. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proc. The 13th ACM Symposium on Operating Systems Principles*, Operating Systems Review, pages 122–136, Pacific Grove CA (USA), October 1991.

[30] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An Operating Sytem Architecture for Application-Level Resource Management. In *Proc. 15th Symposium on Operating System Principles (SOSP-95)*, pages 251–266, 1995.

[31] Dawson Engler. *The Exokernel Operating System Architecture.* PhD thesis, Massachusetts Institute of Technology, October 1993.

[32] J. Liedtke et al. Achieved IPC Performance. In *Proc. 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, May 1998.

[33] Saltzer et al. Introduction to MULTICS. Technical Report MIT/LCS/TR-123, Massachusetts Institute of Technology, February 1974.

[34] D. Floreano, J.-D. Nicoud, and F. Mondada. *Advances in Artificial Life*, volume 1674. Springer_Verlag, 1999. Proc. 5th Euro. Conf. ECAL '99.

[35] B. Ford and J. Lepreau. Microkernels Should Support Passive Objects. In *Proc. Internation Workshop on Object Oriented Operating Systems*, 1993.

[36] Bryan Ford and Erich Stefan Boleyn. Multiboot Standard. http://www.nilo.org/multiboot.html.

[37] Free Software Foundation. GNU's Not UNIX — The GNU project and the Free Software Foundation. http://www.gnu.org/.

[38] E. Gabber, J. Bruno, J. Brustoloni, A. Silberscatz, and C. Small. The Pebble Component-Based Operating System. In *Proc. 1999 USENIX Technical Conference, Monterey, CA*, June 1999.

[39] Bill Gallmeister. *POSIX.4: Programming for the Real World.* O'Reilly & Associates, Inc., January 1995. ISBN: 1-56592-074-0.

[40] Michael Greenwald and David Cheriton. The Synergy Between Non-blocking Synchronization and Operating System Structure. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

[41] Object Management Group. CORBA — Common Object Request Broker Architecture. Technical report, OMG, 1999. http://www.omg.org/.

[42] Graham Hamilton and Sanjay Radia. Using Interface Inheritance to Address Problems in System Software Evolution. *ACM SIGPLAN Notices*, 29(8):119–128, August 1994.

[43] Steven M. Hand. Self-Paging in the Nemesis Operating System. In *Proceedings of the Third Symposium on Operatin Systems Design and Implementation*, February 1999.

[44] Norman Hardy. The KeyKOS Architecture. *Operating Systems Review*, pages pp 8–25, October 1985.

[45] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of μ-Kernel-based Systems. In *Proceedings of the 16th Symposium on Operating Systems*, pages 66–77, 1997.

[46] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kauffmann, 1990. ISBN: 1-55860-329-8.

[47] Dan Hildebrand. An Architectural Overview of QNX. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures: 27–28 April, 1992, Seattle, WA, USA*, pages 113–126, Berkeley, CA, USA, April 1992. ISBN: 1-880446-42-1.

[48] David Hopwood. A Comparison between Java and ActiveX Security. *Proceedings of Compsec '97 — the 14th world Conference on Computer Security, Audit and Control*, 1997.

[49] Wilson Hsieh, Marc Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian Bershad. Compiler Support for System Software. In *Workshop on Compiler Support for System Software*, February 1996.

[50] R. N. Ibbett and P. C. Capon. The Development of the MU5 Computer System. *Communications of the ACM*, 21(1):13–24, January 1978.

[51] Free Software Foundation Inc. GNU Hurd, 1999. http://www.gnu.org/software/hurd/.

[52] Intel. *80386 Hardware Reference Manual.* Intel Corporation, 1987. ISBN: 1-55512-069-5.

[53] Intel Corporation. *Intel Architecture Optimisation Manual*, 1999. Order No. 242816-003.

[54] G. Steele J. Gosling, B. Joy. *The Java Language Specification.* Addison-Wesley, 1996.

[55] Trent Jaeger, Jochen Liedtke, Vsevolod Panteleenko, Yoonho Park, and Nayeem Islam. Security Architecture for Component-Based Operating Systems. In *ACM Special Interest Group in Operating Systems (SIGOPS) European Workshop*, 1998.

[56] E. D. Jensen and J. D. Northcutt. Alpha: a Non-Proprietary Operating System for Mission-Critical Real-Time Distributed Systems. In *Proc. Workshop on Experimental Distributed Systems*. IEEE Computer Society Press, 1990.

[57] M Johnson, K Balasubramanian, and D Johnson. Linux Memory Management Overview, 1992–1996. http://www.linuxdoc.org/LDP/khg/HyperNews/get/memory/linuxmm.html.

[58] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 52–65. ACM Press, October 5–8 1997.

[59] Gerry Kane. *PA-RISC 2.0 Architecture*. Prentice Hall, 1995. ISBN: 0-13-182734-0.

[60] J. Keedy. Paging and Small Segments: A Memory Management Model. In *Proc. 8th World Computer Congress, Melbourne*, 1980.

[61] J. Keedy and J. Rosenberg. Support for Objects in the MONADS Architecture. In *Proc. Workshop on persistent object systems*, pages 202–213, Newcastle NSW (Australia), January 1989.

[62] Brian W. Kernighan and John R. Mashey. The UNIX Programming Environment. *Software – Practice and Experience*, 9(1), January 1979.

[63] Brian W. Kernighan and Dennis M. Ritchie. *The C Progamming Language*. Prentice Hall, 1 edition, 1978. ISBN: 0-13-110163-3.

[64] D. E. Knuth. Structured Programming with Go To Statements. *Computing Surveys*, 6:261–301, 1974.

[65] Jean-Claude Laprie, Jean Arlat, Christian Beounes, and Karama Kanoun. Definition and Analysis of Hardware- and Software-Fault-Tolerance Architectures. *Computer*, 23(7), July 1990.

[66] Greg Law and Julie McCann. 26-Bit Selectors on IA32. In *3rd ECOOP Workshop on Object Orientation and Operating Systems*, June 2000.

[67] Greg Law and Julie McCann. Decomposition of Preemptive Scheduling on the Go! Component-Based Operating System. In *Proceedings of the ACM SIGOPS European Workshop*, September 2000.

[68] Greg Law and Julie McCann. A New Protection Model for Component-Based Operating Systems. In *Proceedings of the IEEE Conference on Computing and Comminications, Phoenix, AZ, USA*, February 2000.

[69] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. In *IEEE Journal on Selected Areas in Communication*, pages 5(1):30–51, September 1996.

[70] H. M. Levy. *Capability-Based Computer Systems.* Digital Press, 1984.

[71] Rick McGowan. The Rhapsody Platform. In *Thirteenth International Unicode Conference: Software + the Internet: Going Global with Unicode (R), September 8–11, 1998, San Jose, California.* The Unicode Consortium, 1998.

[72] Dr. Gary McGraw and Professor Ed Felten. *Java Security: Hostile Applets, Holes and Antidotes.* John Wiley and Sons, 1997. ISBN: 0-4711-7842-X.

[73] Gary McGraw. Java 2's Verifier becomes confused by German student's security attack. *JavaWorld*, April 1999. http://www.javaworld.com/javaworld/jw-04-1999/jw-04-flaw.html.

[74] L. McVoy and C. Staelin. Lmbench: Portable Tools for Performance Analysis. In *Proc. of USENIX Annual Technical Conference*, pages 279–294, 1996. http://www.bitmover.com/lmbench/lmbench-summary.

[75] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales. Aspect-Oriented Programming. *Lecture Notes in Computer Science*, 1357, 1998.

[76] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, February 1997. ISBN: 0-13-629155-4.

[77] Sun Microsystems. HotJava(tm): The Security Story, 1995. http://java.sun.com/sfaq/may95/security.html.

[78] D. Mosberger. *Scout: A Path-Based Operating System.* PhD thesis, University of Arizona, Department of Computer Science, 1997.

[79] S. J. Mullender and G. van Rossum. Amoeba: A Distributed Operating System for the 1990s. In *Distributed Computing Systems: Concepts and Structures*, pages 201–212. IEEE Computer Society Press, 1992.

[80] George C. Necula and Pete Lee. Proof-Carrying Code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[81] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243. USENIX, October 1996.

[82] B. J. Nelson. *Remote Procedure Call.* PhD thesis, Computer Science Department, Carnegie-Mellon University, May 1981. also published as Xerox PARC Technical Report CSL-81-9.

[83] Greg Nelson. *Systems Programming with Modula 3.* Prentice Hall, 1991. ISBN: 0-13-590464-1.

[84] Tim O'Reilly. *BSD Unix.* Addison-Wesley, 1990.

[85] E. I. Organick. *A Programmer's View of the Intel 432 System.* McGraw-Hill, 1983. ISBN: 0–07–047719–1.

[86] John K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proc. 1990 Summer USENIX Conf.*, Anaheim, June 11-15 1990.

[87] S. Radia, G. Hamilton, P. Kessler, and M. Powell. The Spring Object Model. In *Proc. USENIX Conf. on Object-Oriented Technologies*, Monterey CA (USA), June 1995.

[88] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alesandro Forin, David Golub, and Michael B. Jones. Mach: A System Software Kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, February 1989. IEEE Comput. Soc. Press.

[89] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[90] F. Rubin. "GOTO Considered Harmful" Considered Harmful. *Communications of the ACM*, 30(3):195–196, March 1987.

[91] P. Rutter. Uniform Handling of Exceptions in a Stack-Based Language. *ACM SIGPLAN Notices*, 12(9):71–76, September 1977.

[92] S. Schönberg. L4 on Alpha, Design and Implementation. Technical Report CS-TR-407, University of Cambridge, 1996.

[93] M. I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating System Principles*, December 1995.

[94] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a Fast Capability System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 170–185, December 1999. http://www.eros-os.org/.

[95] Jonathan S. Shapiro and Sammuel Webber. Verifying the EROS Confinement Mechanism. In *IEEE Symposium on Security and Privacy*, 2000.

[96] Oliver Spatscheck and Larry L. Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the Third Symposium on Operatin Systems Design and Implementation*, February 1999.

[97] G. L. Steele. Debunking the Expensive Procedure Call Myth. In *Proc. Annual Conference of the ACM*, pages 153–162, 1977.

[98] Helmut G. Stiegler. A Structure for Access Control Lists. *Software, Practice and Experience*, 9(10):813–819, October 1979.

[99] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability — A Study of Field Failures in Operating Systems. In *Proceedings of 21st Internation Symposium on Fault-Tolerant Computing*, June 1991.

[100] Symbolics. MS-DOS Reference Guide. Technical Report 99 47 10, Symbolics, Inc., Cambridge, MA, February 1988.

[101] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van. Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–64, December 1990.

[102] Martin Tasker. EPOC: Core Overview, 1999. http://www.symbian.com/epoc/.

[103] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) Specification, Version 1.2*, 1995. http://x86.org/ftp/manauls/tools/elf.pdf.

[104] Nayeem Islam Trent Jaeger, Jochen Liedtke. Fine-Grained Protection in Operating Systems. In *USENIX Security Symposium.*, 1998.

[105] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungs problem. *Proceedings of the London Mathematical Society*, Series 2(42):230–265, 1936-1937.

[106] Carnegie Mellon University and the Software Engineering Institute in Pittsburgh. *A Practitioner's Handbook for Real-Time Analysis: Guide to RMA for Real-Time Systems.* Kluwer Academic Publishers, 1993. ISBN: 0-7923-9361-9.

[107] Hans van Vilet. *Software Engineering Principles and Practise.* Wiley, 1993. ISBN: 0-471-93611-1.

[108] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, 1993.

[109] Dan Wallach, Dirk Balfanz, Ed Felten, and et al. *Extensible Security Architectures for Java.* Princeton, 1996.

[110] Bruce F. Webster. *The NeXT Book.* Addison-Wesley, 1989. ISBN: 0-201-15851-X.

[111] R. P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[112] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space. In *Proceedings of the Summer 1993 USENIX Conference: June 21–25, 1993, Cincinnati, Ohio, USA.* USENIX, 1993. 1-880446-50-2.

Appendix A

Go! ORB Specification

This appendix gives a full specification of the Go! ORB, described generally in Chapter 6. The ORB's behaviour is broken into 7 categories: Booting; Type and Instance Management; Inter-Component Method Invocation; Linear Space Management; Stack Manipulation; Component Interrogation; and ORB Protection Faults.

A.1 Booting

Go! is Multiboot [36] compliant. The Multiboot standard specifies that the operating system will be presented to the boot-loader as an ELF [103] or a raw binary image. The boot-loader boots the Intel-80386-based PC, puts it into protected mode, creates a flat 32-bit memory model, and transfers control to the image's entry point. In Go!'s case, this entry point is the ORB's constructor. This environment required by the constructor is essentially the Multiboot-defined one (that is, flat mode), except that the ORB's data segment is modified to fit the image precisely.

The ORB is built in ELF format, with a single library OS component[1] at the end of the ELF image's data section. The library OS component is analogous to the UNIX `init` process [4].

The ORB's constructor first sets up the segment-descriptor table, mapping its own code and data segments. The constructor then locates the library-OS component image, installs the type and creates a single instance of it. Once the library-OS component type is installed and a single of instance of it created, the library OS's first method is called. When the this method returns, this indicates that the Go! session has finished, and the system is halted.

[1] In reality this library OS component is likely to contain several others in its data section. See Appendix B.1 for an example.

The Multiboot standard specifies that interrupts are disabled when control is transferred from the boot-loader to the kernel. Go! will not re-enable interrupts (that is, interrupts are guaranteed to be disabled when the library operating system is called).

A.1.1 Specifications

`void ctor()` `ctor` is called by jumping to the first byte of the ORB's code section. The processor must be in 32-bit protected mode, as specified by the Multiboot standard, except that the code and data segments must fit the ORB image's code and data sections exactly (the data section includes the image of the single library-OS component appended to the end).

Algorithm

- build temporary GDT that maps ORB's code and data segments exactly
- examine processor state to find location and size of both the ORB and library operating system images
- move the library operating system image in memory to make room for ORB's uninitialised data (BSS)
- initialise 'Global-Descriptor Table' and load ORB's code and data segments
- display welcome message
- create a `stack` component to represent current stack
- initialise ORB free-lists (CDT, GDT, stack descriptor and private-`t-call`-stack)
- initialise CDT entries
- install library-operating-system component type
- create an instance of library-OS component, and invoke the first method
- display 'good-bye' message, and shutdown the system

A.2 Type and Instance Management

Before a component of a given type can be created, that type must be registered with the system — that is, *installed.* Only implementation types are installed — if required, interface types are managed by the library OS. To install a new component type, the ORB's `install` method is called,

and a null component passed as the single parameter. The `null` component passed contains the image of a new type to install at the end which specifies locations of the image's various sections. The ORB registers the interface and implementation types along with the code segment and initial-data segment.

It is also possible to *uninstall* types after their use. The `uninstall` method is provided to de-register types from the ORB.

A.2.1 Specifications

```
objref install( objref image )
```

`install` will install a new type. The method is invoked by issuing the instruction `call 8:56`, with `image` in register `ebx`. Only the `esp` and `ebp` registers are preserved. `image` should be the ObjRef of a `null` component, that specifies a component implementation as contiguous sections of text, data, BSS and a method table. Appended to the end of the image is a table that describes these sections' locations. This table looks like (in C++)

```
struct {
   struct section {
      unsigned start;
      unsigned size;
   };

   section text;
   section init_data;
   section BSS;
   section method_table;
};
```

The method table should be in the following format:

```
struct mt {

   struct mt_entry {
      void *start;
      unsigned pad;
   };

   unsigned method_count;
   unsigned pad1;
   void*    ctor_entry;
   unsigned pad2;
   void*    dtor_entry;
   unsigned pad3;

   mt_entry table[0];
```

};

where the mt_table field contains method_count mt_entry structures, one for each method entry point.

If installation succeeds, the null-component specified by image is destroyed, and the new type is referred to by the ObjRef image. The ObjRef of the new type's initial-data segment is returned in eax. The new type is installed 'in place' — that is, the library operating system should not allocate any new memory, or reclaim the memory from the null component (with the exception of the sections descriptor at the end of the image which can be freed).

Algorithm

- switch to ORB 'dynamic' data segment
- 'destroy' the null-component referred to by image by writing FFFFh to its sel field in the CDT.
- allocate a new ObjRef for the initial-data segment from the ORB's CDT free-list (the compare-and-exchange instruction is used to allocate the reference in a non-blocking and thread-safe manner)
- modify image's associated segment to become a code segment, and modify its base and limit to map the image's text section
- use the compare-and-exchange operation to allocate a new ObjRef from the CDT free-list in a thread-safe, but non-blocking manner
- map the new ObjRef's data segment to reference the initial data (that is, so that its base is the sum of image's base and data, and its size is mt less data)
- set the new ObjRef's mtable entry in the CDT to bss
- set image's type_ref field in the CDT to the new ObjRef
- set image's mtable field in the CDT to method table's offset from the ORB's data segment (that is, the sum of image's segment's base and mt, less the ORB's data segment base)
- mark the new type as valid
- switch back to caller's data segment, and return the new ObjRef

```
void uninstall( objref type )
```

`uninstall` will de-register the type identified by `type` from the system. `uninstall` is invoked by placing `type` in the register `ebx`, and issuing the instruction `call 8:64`. All registers apart from `ebx` and `eax` are preserved.

It is assumed `uninstall` is only called once there are no instances of that type (that is, it is up to the library operating system to ensure types aren't uninstalled while instances still exists). This policy of delegating synchronisation allows the library operating system to specify the policy (for example, reference counts) as well as easing the task for managing the linear memory associated with the type.

Algorithm

- switch to ORB 'dynamic' data segment
- invalidate the type by setting the `call_count` field in `type`'s CDT entry to 0
- take a GDT lock on `type`'s code and initial-data segments
- if they're cached in the GDT, free up the selectors
- release the GDT locks in `type`'s code and initial-data segments
- free `type`'s code and initial-data ObjRefs
- switch back to caller's data segment
- return by issuing the inter-segment return instruction

```
objref create( uint count, objref type, uint linear, uint para_sz, ... )
```

Calling `create` with `count` of 1 will create an instance of the type `type` at linear address `linear`, and calls its constructor, passing `para_sz` bytes of parameters which follow on the stack. To invoke `create` with a `count` of 1, place 1 in register `eax` and push `type`, `linear` and `param_sz` onto the stack, followed by any parameters (parameters to the constructor and `create` should be pushed in the reverse order that they appear in the signature). Finally, issue the instruction `call 8:40`. On return, `eax` contains an ObjRef that identifies the new instance. All other registers are preserved.

`create` can be used to create several instances with adjacent ObjRefs, by calling with `count` greater than one. In this case the `type`, `linear` , `param_sz` and constructor parameters should be

pushed for each instance to create in turn. Instances are created in the opposite order to that in which they are pushed onto the stack. The ObjRefs of each new instance are contiguous, starting from the one returned in register `eax`.

Algorithm

- switch to ORB's 'dynamic' data segment
- allocate `count` *consecutive* ObjRefs (leaving the `sel` fields as `FFFFh`)
- while `count` is greater than zero:
  - create a segment referencing the new component's location in linear memory as specified by `linear` and of the size specified by `type`
  - associate the ObjRef with the new segment and `type`, but set the `mcount` field to zero
  - copy `type`'s initial-data segment into the newly-allocated data segment
  - take a GDT lock on the new ObjRef
  - allocate a new selector in the GDT for ObjRef
  - copy the constructor's parameters 24 bytes beyond the current stack-pointer (for use with `call`)
  - if `type` specifies a `stack`, call the stack constructor (see Section A.5.1), otherwise if `type` does not specify a `null` component, `call` the constructor on the new ObjRef
  - set the new ObjRef's `mcount` field (thus allowing methods to be called)
  - release the GDT-lock on the new ObjRef
  - decrement `count`
- switch to caller's data segment, and return the first of the new ObjRefs in `eax`

`void destroy( objref to_go, uint count )`

`destroy` destroys `count` components identified by consecutive references starting from `to_go`, and frees resources held by the ORB associated with `to_go` (note that this does not include linear space, which is managed by the library operating system). To invoke `destroy`, place `to_go` in register `ebx` and `count` in `eax`, and issue the instruction `call 8:48`. `eax` is 'clobbered', but all other registers are preserved.

If there are any returns pending or GDT locks held on the component it becomes a zombie (see Section 6.3.2). However, even if there are GDT locks held, the selector associated with `to_go` is still freed — it is up to the library operating system to ensure that if a component's selector must remain valid then it is not destroyed.

Algorithm

- switch to ORB's dynamic data segment
- for each of the `count` references to destroy:
  - use the compare-and-exchange operation to mark `to_go` as invalid by atomically setting its `sel` field in the CDT to `FFFFh` only if it is a valid component instance
  - take a GDT-lock on `to_go`
  - call the destructor
  - release the data segment from the GDT by placing the selector in the GDT free-list
  - if the `lock_count` and `call_count` are zero, there are no pending `returns`: release the ObjRef by adding it to the CDT free-list[2], otherwise mark the ObjRef as a zombie
- switch back to the caller's data segment and issue an inter-segment return instruction

A.3 Inter-Component Method Invocation

An overview of inter-component method invocation was presented in Section 5.4. The various 'flavours' of `call` and `return` have also been outlined in 6.3.1. This section explains in detail how the Go! ORB implements the different flavours of `call` and `return`.

A.3.1 Specifications

```
any call( objref callee, uint method )
```

`call` issues a method-call on method number `method` on the component identified by `callee`. To invoke, place `callee` in register `ebx` and `method` in register `ecx` before issuing the instruction `call 8:8`. Only register `esp` is guaranteed to be preserved, even between control leaving the caller and arriving at the callee (that is, the ORB will clobber registers). The ORB will also clobber the

[2]Immediate re-use of ObjRefs may present a problem in certain systems. In this case, a 'death-row' free-list should be maintained to force a period between an ObjRef being freed and reused

next 20 bytes of the stack: any arguments should be placed starting from 24 bytes beyond the stack-pointer. Control is guaranteed to return to the next instruction of the caller component, but no bounds are given on the time this takes. Note that at entry to the callee method `esp` will have been incremented by 24, ready for the callee to read these arguments.

If `callee` does not specify a valid component an `xcp_orb_noref` exception is thrown. If `method` is out of range of the methods in `callee`'s interface, an `xcp_orb_invalid` exception is thrown.

Algorithm

- load ORB data-segment selector into the data-segment register
- push caller's reference onto the stack
- increment callee's call-count
- validate callee method number is within callee's method count
- push details of previous call onto the stack (e.g. previous stack size)
- shrink stack by manipulating segment-descriptor tables
- increment call-depth associated with current stack segment
- look-up target code segment and offset in method table
- load callee data segment into the data-segment register via the CDT
- place caller ObjRef in general-purpose register (authenticate)
- jump into callee code segment at offset indicated by method table

```
any f-call( objref callee, uint method )
```

`f-call` issues an `f-call` on method number `method` on the component identified by `callee`. To invoke, place `callee` in register `ebx` and `method` in register `ecx` before issuing the instruction `call 8:16`. Only register `esp` is guaranteed to be preserved, even between control leaving the caller and arriving at the callee (that is, the ORB will clobber registers). Stacks are changed to the next stack in the stack-chain before control enters `callee` — if the callee requires parameters these must be placed on the next stack in the stack-chain[3].

[3]This can be done by having the library OS arranging stacks in a stack chain to overlap the same linear memory, and caller retrieving from or specifying to the library OS where the next stack 'starts'.

If there is no next stack in the current stack-chain, an `xcp_orb_nostack` exception is thrown. If `callee` does not specify a valid component an `xcp_orb_noref` exception is thrown. If `method` is out of range of the methods in `callee`'s interface, an `xcp_orb_invalid` exception is thrown.

Control is guaranteed to return to the caller's next instruction, but no bounds are given on the time this takes.

Algorithm

- load ORB data-segment selector into data-segment register
- push caller's ObjRef onto the stack
- increment callee's call-count
- validate method number is within callee's method count
- push details of previous call onto the stack (e.g. previous call-depth)
- find the 'next' stack in chain
- switch to the next stack in chain
- place caller's ObjRef into a general-purpose register (authenticate)
- look-up callee's method table
- load callee's data segment into data segment register via the CDT
- jump to callee's entry point

`any t-call( objref callee, uint method )`

`t-call` issues a `t-call` on method number `method` on the component identified by `callee`. To invoke, place `callee` in register `ebx` and `method` in register `ecx` before issuing the instruction `call 8:24`. Only register `esp` is guaranteed to be preserved, even between control leaving the caller and arriving at the callee (that is, the ORB will clobber registers). The stack is not protected (that is, the callee has full access to the caller's stack frame). However, control is guaranteed to return to the next instruction, although no bounds are given on the time this takes.

If `callee` does not specify a valid component an `xcp_orb_noref` exception is thrown. If `method` is out of range of the methods in `callee`'s interface, an `xcp_orb_invalid` exception is thrown.

Algorithm

- load ORB data-segment selector into data-segment register
- push caller's ObjRef onto the stack
- increment callee component's call-count
- validate caller ObjRef and method number
- obtain element for 't-call list' to record return information
- record return information and link element into the stack's t-call list
- increment callee's call count
- look-up callee's method table
- place caller ObjRef in general-purpose register (authenticate)
- load callee's data segment into data-segment register via the CDT
- jump to callee's entry point

```
noreturn xfer( objref callee, uint method )
```

`xfer` performs a control transfer to method number `method` on the component identified by `callee`. To invoke, place `callee` in register `ebx` and `method` in register `ecx` before issuing the instruction `call 8:32`. Only register `esp` is guaranteed to be preserved, even between control leaving the caller and arriving at the callee (that is, the ORB will clobber registers). The stack is not protected (that is, the callee has full access to the caller's stack frame). Control will not return — if the callee issues a `return`, control will return as if the caller had issued it.

If `callee` does not specify a valid component an `xcp_orb_noref` exception is thrown. If `method` is out of range of the methods in `callee`'s interface, an `xcp_orb_invalid` exception is thrown.

Algorithm

- load ORB's static-data-segment selector into the data-segment register
- validate method number within callee's method count
- decrement caller component's call-count

- increment callee component's call-count
- look up callee's method table
- place caller's ObjRef in general-purpose register (authenticate)
- load to callee's data segment into data-segment register via the CDT
- jump to callee's entry point

`noreturn return()`

`return` will cause control to return to the instruction following the most recent `call`, `f-call` or `t-call` of the current stack. Registers `eax` and `edx` are preserved so that they may be used for return values. `esp` is set to its value at the corresponding `call`, `f-call` or `t-call`. No other registers are preserved. To invoke a `return`, issue the instruction `jmp 8:176`. If there are no `returns` pending on the current stack an `xcp_orb_noret` exception is thrown.

Algorithm

- load ORB 'dynamic' data segment
- if current stack's `call-depth` is zero, return from an `f-call` by:
  - find 'previous' stack in chain
  - switch to 'previous' stack in chain
  - pop details of previous call from stack (e.g. previous stack size)
  - decrement callee's call-count
  - pop callee's ObjRef from the stack, and load data segment via the CDT
  - issue inter-segment return to caller
- if the t-call-list is empty, return from a `call` by:
  - load ORB data-segment selector into the data-segment register
  - determine type of return (assuming return from `call`)
  - re-grow stack by manipulating segment-descriptor tables
  - pop details of previous call from stack (e.g. previous stack size)

- decrement call-depth associated with current stack segment
- decrement callee's call-count
- pop caller's ObjRef from the stack and load data segment via the CDT
- issue an inter-segment return, returning control to caller

- otherwise, return from `t-call` by:
  - load ORB's static-data-segment selector into the data-segment register
  - determine type of return (assume return from `t-call`)
  - restore the return information stored in first element from 't-call list'
  - free the return element to ORB's internal memory
  - decrement callee's call-count
  - pop callee's ObjRef from the stack, and load data segment via the CDT
  - issue inter-segment return to callee

`noreturn throw( objref xcp )`

This method throws an exception. To throw an exception, the ObjRef `xcp` is placed in register `eax`, and `jmp 8:168` is issued. The stack is unwound and the caller[4] ObjRef, then the offset that control would have returned to, and the component that issued the exception are placed in general-purpose registers `eax`, `ebx` and `ecx` respectively. Finally an interrupt is triggered on vector 30h.

Algorithm

- load ORB 'dynamic' data segment
- unwind call-stack (as in `return`)
- place return target ObjRef (callee) and return target offset in general-purpose registers
- generate a software interrupt on vector `30h`

[4]The 'caller' is the component to which control would have returned if a `return` was issued.

A.4 Linear-Space Management

This section documents the ORB methods concerned with the management of linear space. An introduction to Go!'s linear-space management is given in Section 6.3.4.

A.4.1 Specifications

`uint linear( bool grant, uint free_list )`

`linear` is called in order to provide linear space to, or revoke linear space from, the ORB. It is called by issuing the instruction `call 8:136`, with the `grant` parameter in register `ebx`, and `free_list` in `ecx`. The return value is placed in register `eax`. All other registers are preserved.

If the `grant` parameter has the value `true`, then the ORB claims the linear space specified by `free_list` for its internal data structures. Otherwise the ORB relinquishes the linear space specified by a returned free list. The return value (or `free_list` if `grant` is `true`) is the linear address of a block of linear memory. The first 4 bytes of the free block give its size, the second give the linear address of another such block (an address of `FFFFFFFFh` indicates that there are no more free blocks). It is anticipated that library operating systems will call `linear` with `grant` equal to `true` in response to `orb_xcp_nomem` exceptions, and with `grant` equal to `false` when the library operating system is running out of free memory. If `linear` is being used to pass linear space to the ORB in response to an `xcp_orb_nomem` exception, the blocks should be the same size as that returned by the exception's `info` method. The free-list returned (even when `linear` was called with `grant` equal to `true`) are not being used by the ORB, and should be reclaimed by the memory manager to avoid leaks — that is, when memory is given to the ORB with `linear`, the ORB is free to return blocks it cannot use or does not require.

Algorithm

- switch to ORB 'dynamic' data segment
- set the return value to `FFFFFFFFh`
- if `grant` is equal to `true`, for each element in `free_list`:
  - if the element is not 16 or 32 bytes, set the second 4 bytes of the block to the return value, and set the return value to the linear address of this block.

- otherwise, use the atomic compare-and-exchange instruction to add the block to the relevant free list in a thread-safe, but non-blocking manner

- if `grant` is equal to `false`, for both the 16 and 32 byte free-lists, while they are not empty:
  - use the compare-and-exchange operation to remove the first element of the free-list in a thread-safe but non-blocking manner
  - set the second 4 bytes of the removed block to the return value, and set the return-value to the linear address of the removed block

- switch back to the caller's data segment and return the return-value

A.5 Stack Manipulation

As mentioned in Section 6.3.3, Go! defines the `stack` base component type. If `create` is called to create a stack type, instead of the normal component creation, a "stack descriptor" is allocated, defining:

`next_esp` stores the incoming stack pointer value of the next stack to be loading during `f-call` (that is, the next stack's data-segment size)

`next_ss` the stack segment of the next stack in the `f-call` stack chain

`prev_ss` the stack segment of the previous stack in the `f-call` stack chain

`prev_esp` used to store away the outgoing stack pointer for outgoing stacks during `f-call`

`next_ref` the ObjRef of the next stack in the `f-call` stack chain

`prev_ref` the ObjRef of the previous stack in the `f-call` stack chain

`call_depth` the number of `calls` issued on *this* stack (that is, since the last `f-call`)

`private` the offset within the ORB's 'dynamic' data segment of this stack's 'private call stack': a linked list of elements containing return information for `t-calls`

`this_ref` this stack's ObjRef

`limit` the size of this stack's data segment

`prev_ptr` a pointer to the previous stack descriptor

`next_ptr` a pointer to the next stack descriptor

`nomem_xcp` an ObjRef of a pre-created `xcp_orb_nomem` exception (to avoid dead-lock when running out of memory)

The `stack` type's constructor takes an ObjRef argument. This should be either zero or another stack type to which this stack will be attached in order to form a stack-chain for use with `f-call`.

It is anticipated that instances of the `stack` type will be created by library operating systems as part of thread creation, and (indirectly) by components preparing to use `f-call`. The library-operating-system scheduler will need to call `switch_stack` to set up the new stack segment when new threads are scheduled. This changes the ORB stack specific data (such as `call_depth`).

Unlike all other ORB methods, `switch_stack` **and** `stack`**'s constructor are not re-entrant**.

Note that the various call primitives require access to these structures, but they are dynamic data. As mentioned in Section 6.5.1, this is undesirable: call primitives are restricted to accessing static data only. To overcome this, during `stack_switch`, the incoming stack-chain is copied to buffers in the ORB's 'static' data segment.

A.5.1 Specifications

```
void stack::ctor( uint size, objref link_to )
```

The 'stack constructor' is called if `create` is invoked specifying a type of `stack`. Once the `create` completes, the new component may be used as a target to `switch_stack`.

`stack`**'s constructor is not re-entrant**. The library operating system must prevent concurrent attempts to attach new stacks to a stack-chain.

Algorithm

- switch to ORB's 'dynamic' data segment
- claim a stack descriptor from the ORB's `stack_descriptor` free-list
- overwrite this component's `mtable` field in the CDT to contain a pointer to the new stack descriptor
- set the new descriptor's `next_ref`, `call_depth` and `private` fields to zero
- set the new descriptor's `next_ss` field to `FFFFh`

- if `link_to` is non-zero, link the new stack into the chain by:
  - set `link_to`'s `new_esp` to `size`
  - set `link_to`'s `next_ss` to `comp`'s selector from CDT
  - set `link_to`'s `next_ref` to the new stack component's ObjRef
  - set the new descriptor's `prev_ref` to `link_to`
  - set the new descriptor's `prev_ss` to `link_to`'s `this_ss`
- switch back to the caller's data segment, and return using an inter-segment return

`void switch_stack( comp new_stack )`

The `switch_stack` method is called to switch the active stack. It is anticipated that this will be done as part of thread scheduling by the library operating system. The method is invoked by issuing the instruction `call 8:160` with the ObjRef of the new stack in `ebx`.

Note: `switch_stack` is not re-entrant. The library operating system must prevent concurrent calls to `switch_stacks`.

Algorithm

- switch to the ORB's 'static' data segment
- set the ORB's `current_stack` pointer to `new_stack`
- switch back to the caller's data segment, and return using an inter-segment return

`objref get_stack( objref chain, uint walk )`

Returns `walk` elements along the stack-chain from stack `chain`. If `chain` is passed as zero the current stack is used.

Algorithm

- if `first` is zero, set `chain` to the currently active stack
- while `walk` is non-zero:
  - set `chain` to the next stack from `chain` in the stack-chain
  - decrement `walk`
- switch to caller's data segment and return `chain`

A.6 Component Interrogation/Manipulation

A.6.1 Specifications

`uint lock( objref ref )`

The segment associated with `ref` is locked in the GDT at least until a subsequent call is made to `unlock( ref )`. If locking `ref` in the GDT is not possible (due to the GDT being full of locked-in segments) an `xcp_orb_gdtfull` exception will be thrown. Concurrent calls to `lock` and `unlock` for the same ObjRef are supported for up to 2^{32} concurrent attempts — that is, `lock` increments a count which `unlock` decrements. The selector of the ObjRef's segment is returned.

To call `lock`, place `ref` in register `ebx` and issue the instruction `call 8:72`. The result is returned in register `eax`. All other registers are preserved.

Algorithm

- switch to the ORB's static data segment
- increment the ObjRef's lock count
- cache the ObjRef in the GDT if not so already
- switch back to caller's data segment
- return the GDT selector of the ObjRef

`void unlock( objref ref )`

The segment associated with `ref` is unlocked from the GDT, and so may be rejected sometime in the future. Concurrent calls to `lock` and `unlock` for the same ObjRef are supported for up to 2^{32} concurrent attempts — that is, `lock` increments a count which `unlock` decrements.

To call `unlock`, place `ref` in register `ebx` and issue the instruction `call 8:80`. All registers other than `eax` are preserved.

- switch to the ORB's static data segment
- decrement the ObjRef's lock count
- cache the ObjRef in the GDT if not so already
- switch back to caller's data segment

- return the GDT selector of the ObjRef

`objref sel2ref( uint16_t s )`

This method returns the ObjRef that was associated with selector `s` at the time of calling. It is called by placing the selector `s` in register `ebx` and issuing the instruction `call 8:96`. The associated ObjRef is returned in register `eax`. All other registers are preserved.

Note that to use safely whatever component is associated with selector `s` should be known to be locked in the GDT (otherwise a race occurs between the return value and the ObjRef being rejected from the GDT).

Algorithm

- switch to ORB's 'static' data segment
- look up the selector's associated reference from the 'reverse' CDT, and place in `eax`
- switch back to caller's data segment and issue an inter-segment return

`objref get_self()`

A component can determine its own reference by calling the `get_self` ORB method. This is the analogue of the UNIX `get_pid` system call. This method simply returns the ObjRef of the caller. It is called by issuing the instruction `call 8:104`. The objref is returned in register `eax`. All other registers are preserved.

Algorithm

- take a local copy of the data-segment register
- switch to ORB's 'static' data segment
- place the caller's ObjRef in register `eax`
- switch back to caller's data segment and issue an inter-segment return

`objref get_type( objref comp )`

This method simply returns the ObjRef identifying `comp`'s type. It is called by issuing the instruction `call 8:112`, and placing `comp` in register `ebx`. The objref is returned in register `eax`. All other registers are preserved.

Algorithm

- switch to ORB's 'static' data segment
- validate that `comp` is a valid instance ObjRef (otherwise throw an `xcp_orb_invalid` exception)
- place `comp`'s `type_ref` field from the its entry in the CDT into register `eax`
- switch back to caller's data segment and issue an inter-segment return

`objref set_type( objref comp, uint new_type, uint linear, uint force )`

This method switches the ObjRef identifying `comp`'s type. If the call succeeds, the outgoing implementation's destructor is called, before the incoming's constructor. The new implementation's data segment is located at linear address `linear`.

If `force` contains 2 then the type is changed, even if there are returns pending on some of the implementation's methods. If `force` contains the value 0 or 1 then the swap only happens if there are no threads pending on the implementation. If `force` contains 1, all future calls onto the implementation on this ObjRef are blocked by setting its method count to zero. If `force` contains 0, other methods are allowed in as normal.

`set_type` is called by pushing any parameters to pass to `new_type`'s constructor, followed by the size in bytes of these parameters, `new_type`, and `linear` onto the stack. Then placing `comp` in register `ebx`, and `force` in register `ecx` and issuing the instruction `call 8:120`. If the swap succeeds, `comp` is returned in register `eax`, otherwise zero is. All other registers are preserved.

Algorithm

- switch to ORB's 'static' data segment
- if the `force` parameter is not 2:
  - if `comp`'s entry has a non-zero call_count then switch to caller's DS and return zero
  - set `comp`'s method count to zero
  - if `comp`'s entry has a non-zero call_count then:
  - if `force` is equal to 1 then switch to caller's DS and return zero
  - reinstate `comp`'s original method count

– switch to caller's DS and return zero

- call the ORB's destroy method on `comp` after setting `comp`'s `call_count` to 1 so that `comp` will be left as a zombie
- create a new instance of type `new_type` at linear address `linear`, including calling the constructor
- switch back to caller's data segment and issue an inter-segment return

`objref setdesc( objref r, uint or1, uint or2, uint and1, uint and2 )`

This method changes the descriptor associated with reference `r`. A bit-wise AND is performed with the bottom double-word of the descriptor and the `and1` parameter, and the top double-word of the descriptor and `and2`. Similarly, a bit-wise OR is performed between the bottom double-word of the descriptor and `or1` and the top double-word of the descriptor and `or2`. The bottom-half of the original descriptor is returned in register `eax` and the top half in `edx`. All other registers are preserved.

To call `setdesc` place `r` in `ebx`, `or1` in `ecx`, `or2` in `edx`, `and1` in `esi` and `and2` in `edi` and issue the instruction `call 8:128`.

Algorithm

- switch to ORB's 'static' data segment
- perform the bit-wise AND and ORs as specified in the specification, leaving original descriptor in `eax:edx`
- switch back to caller's data segment and issue an inter-segment return

void `objref reject( objref r )`

This method simply rejects ObjRef `r` from the GDT. It is intended for use by the library operating system to optimise rejection policy from the GDT.

To call `reject` simply place the ObjRef to reject in register `ebx` and issue the instruction `call 8:152`. All registers are preserved.

Algorithm

- switch to ORB's 'static' data segment
- if ObjRef `r` is cached in the GDT, reject it
- switch back to caller's data segment and issue an inter-segment return

A.7 ORB Protection Faults

As mentioned in Section 6.3.3, components that invoke ORB methods with invalid parameters will receive exceptions. The ORB can generate hardware-protection faults in four scenarios (not withstanding bugs!):

- The attempted return to a zombie as outlined in section 6.3.2 will mean that the *ORB* faults. This is because the attempted return goes via the ORB, and it is from here that the zombie's data segment is loaded.
- Attempting to call a method on an invalid component reference causes a protection fault. This is a deliberate action, and allows the ORB to omit a potentially expensive conditional branch to speed up `call`.
- Attempting to load a valid component reference, but one that is is not cached in the segment descriptor table (see Section 6.5.1). Again, this avoids potentially expensive conditional branches during RPCs.
- Attempting to `fcall` when there is no next stack in the current stack-chain, or `return` when there is no return pending.
- Calling the ORB without sufficient space remaining on the current stack (a stack-fault will be triggered). In this case, it is up to the library operating system to provide sufficient stack space to complete the operation and restart the faulting instruction. Note that all ORB methods (other than `create`) are guaranteed to require no more than 128 bytes of stack space; most will require much less. `create` will use no more than the largest amount of parameters to send to any constructor plus 32 bytes times the number of components to create.

Whatever component is managing exceptions need not be aware of these details (indeed, they may change for different implementations of the ORB). It is trivial to discover the code segment

at the time of a protection fault. If the library operating system discovers that the faulting code segment was indeed the ORB's (selector 8) then it can conclude that the ORB generated the fault. The ORB's `fault` method must be called in response to the ORB generating a 'general protection fault' (note this does not include faults such as page faults). Arguments to be passed to `fault` should be pushed on to the stack right-most first.

A.7.1 Specifications

```
noreturn fault( uint DS, uint regset[8], uint ec, uint eflags, uint stack[4] )
```

ORB faults must be dealt with by invoking the ORB's `fault` primitive in the context of the faulting thread. `fault` requires several arguments to be pushed on the stack:

`uint DS` The data-segment register at the occurrence of the fault (either the ORB's 'static' or 'dynamic' data segment)

`uint regset[8]` The general-purpose registers in the order pushed by IA32's `pushad` instruction

`uint ec` The 'error code' field stacked by the processor in response to the protection fault

`uint eip` The program counter that faulted

`uint eflags` The `eflags` register at the time of the fault

`uint stack[4]` The top 4 words of the stack at the time of the fault

`fault` is called by issuing the instruction `jmp 8:144`, with the stack set up as above. If the fault is due to the attempted reference of a segment not cached in the GDT, said segment is brought into the GDT, and the faulting instruction retried. If the fault was caused by the attempted use of an invalid ObjRef (either through a method call with invalid parameters, or a return to a zombie), the operation is 'undone' and an `xcp_orb_noref` exception is thrown. If the fault is for some other, undetermined reason, the system is halted, and a diagnostic message displayed.

Algorithm

- load the ORB's 'dynamic' data segment
- examine `eip`: if it denotes that the fault happened when attempting to load the next stack in the current stack chain this indicates that an `f-call` was issued when there are no more stacks: throw an `xcp_orb_nostack` exception

- examine `eip`: if it denotes that the fault happened when attempting to load the previous stack in the current stack chain, this indicates that a `return` was issued when there were no RPC returns pending: throw an `xcp_orb_noret` exception
- examine `eip`. If it denotes that the fault happened when attempting to load any other invalid selector into a segment register:
  - if `regs[2]` (`ebx` at the time of the fault) indexes a CDT entry with selector `FFFFh`, an attempt was made to use an invalid ObjRef: throw an `xcp_orb_noref` exception
  - if `regs[2]` (`ebx` at the time of the fault) indexes a CDT entry with selector `FFFEh`, an attempt was made to reference a valid, but uncached ObjRef. Find a free selector (rejecting eligible segments from the GDT if necessary), copy the ObjRef's descriptor to its entry in the GDT, set the selector field in the CDT to the new selector, and retry the faulting instruction
  - if `regs[2]` (`ebx` at the time of the fault) indexes a CDT entry with type `FFFCh`, a return was made to a zombie component. Throw an exception of type `xcp_orb_zombie`.
- if control reaches here, the ORB has faulted unexpectedly, almost certainly due to a bug or improper use of a privileged method. Display a diagnostic message, and halt the system

Appendix B

GTE Programmer's Manual

This manual gives an overview of GTE — the Go! Test Environment. GTE is not meant as a complete system, simply one to explore the issues of building library operating systems on top of the Go! ORB, and more specifically to complete the proof-of-concept of SISR.

Each GTE component has a section dedicated to it, describing that component in terms of:

Overview A brief description of the component's expected use, its motivation, and its role within GTE.

Interface The IDL of the component's interface, followed by descriptions of each method. Unless stated otherwise, each component's `ctor` method simply initialises it and `dtor` closes it down.

Implementation An overview of the component's implementation is given. This is most useful as a companion to the component's source code.

B.1 `comp_lib`

As described in Section A.1, Go! requires that a single component representing a library operating system is incorporated into the ORB's data section. In GTE's case, this component is `comp_lib`. `comp_lib` contains a simple, read-only, single-directory RAM file-system. The image for this file system resides in `comp_lib`'s data section. The file-system contains the images of all the other components that make up GTE.

B.1.1 Overview

comp_lib inherits from GTE's name_ctx interface. Components can be looked up (and installed) by name, using the lookup method. In effect, it uses the name_ctx type to realise a single-directory, read-only filesystem. Figure B.1 shows how component images are embedded in other component images' data sections.

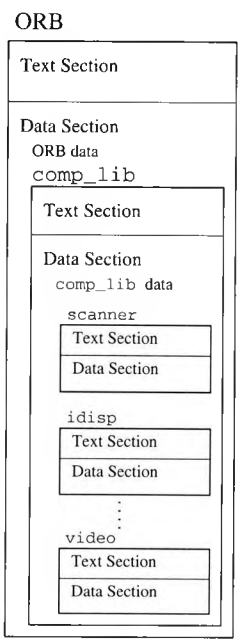


Figure B.1: Component images embedded in other component images' data sections

B.1.2 Interface

```
interface name_ctx {
   void ctor();
   void dtor();

   objref lookup( gstr name );
};

interface comp_lib : name_ctx {
    void handle();
    void null();
};


objref lookup( gstr name )
```

lookup returns a component from the comp_lib that corresponds to name.

A `null` component is returned that contains a component implementation in the format required by Go!'s `install` method (see Section A.2.1). The image returned depends on the name passed as the single parameter. The `gstr` type refers to a 32-bit, unsigned integer representing a string's length pushed onto the stack, followed by an ASCII representation of said string.

If no match is found for the name, an `xcp_namectx_nomatch` exception is thrown.

There is a caveat to this description of `lookup`. The first time `lookup` is called is by the ORB on startup (see Section A.1). This means that rather than performing a `lookup` as above, the first invocation will in fact bring up GTE, installing and creating instances of all the GTE components. All subsequent calls to `lookup` will behave as documented above.

```
void handle()
```

This method is used internally by `comp_lib`. It is used for entry when control is transfered to a fully-fledged GTE thread. If called by any component other than `comp_lib`, an `xcp_gte_prot` exception is returned.

```
void null()
```

Used internally by `comp_lib` to test null-RPC times, this method returns immediately.

B.1.3 Implementation

The `comp_lib` data section consists of a lists of $(name, offset)$ pairs, where *name* is the name of the component, and $offset$ is a pointer to the image within `comp_lib`'s data section. The list is followed by the image of each component in the list.

A tool has been developed called `libgen` that constructs the data section from a directory of component images on the build operating system (currently only tested under Linux).

B.2 mem_mgr

The `mem_mgr` component is responsible for managing linear space. Future versions of `mem_mgr` might include support for demand-paging, or exotic allocation schemes. However, the current version of `mem_mgr` implements only simple, free-list-based allocation/de-allocation of linear memory.

B.2.1 Overview

Go! uses an external component to manage linear memory, as described in Section 6.3.4. `mem_mgr` is GTE's version of that component. Creation/destruction and (un)installation are directed via this component, which selects the linear addresses at which components' data reside.

B.2.2 Interface

```
interface mem_mgr {
   void   ctor();
   void   dtor();

   objref install( objref image );
   void   uninstall( objref type );

   objref create( objref type, uint argc, ... );
   void   destroy( objref instance );

   uint   get_addr( objref c );
};
```

```
void install( objref image )
```

This method is essentially a veneer for the ORB's `install` method. `image` is the ObjRef of a null component with a data segment that describes an implementation type in a proprietary format. The null component referred to by `image` is destroyed as part of the installation. The ObjRef of the new type is returned.

Successful installation of the type is subject to it passing a code scan. The text section of the image described by `image` is scanned prior to installation by calling the `scan` method on `scanner`. Following successful installation of the type, instances may be created using the `create` method.

If `image` is not a valid ObjRef referring to a null-component containing a valid description of a type in its data segment, an `xcp_gte_invalid` exception is thrown. If code-scanning fails (that is, the image's text section contains privileged instructions) an `xcp_gte_unsafe` exception is thrown.

```
void uninstall( objref type )
```

If `type` is not an ObjRef of a type previously `installed` an `xcp_gte_invalid` exception is thrown. Before uninstallation the `release` method is called on `scanner`.

```
void create( objref type, uint argc, ... )
```

If `type` is not an ObjRef referring to a type successfully installed via `install` (and not subsequently `uninstalled`), an `xcp_gte_invalid` exception is thrown. If there is insufficient memory to create an instance of said type, an `xcp_gte_nomem` exception is thrown. Otherwise, an instance is created of the type specified by `type`. `argc` bytes of arguments are passed to the constructor. If successful, an ObjRef referring to the new instance is returned.

```
void destroy( objref instance )
```

If `instance` is not a valid ObjRef referring to a component instantiated via `create`, an `xcp_gte_invalid` exception is thrown. Otherwise, the component referred to by ObjRef `instance` is destroyed, and its memory freed.

Hint: if a component calls `destroy` on itself (thus 'self-destructing'), it can avoid generating an `xcp_orb_zombie` exception by `xfering` to, rather than `calling` `mem_mgr::destroy()`.

```
uint get_addr( objref instance )
```

The linear address at which the component referenced by ObjRef `instance` was created is returned. If `instance` does not refer to a type created by `comp_lib` an `xcp_gte_invalid` exception is thrown.

B.2.3 Implementation

The current version of GTE has very simple, and naive memory management (the purpose of GTE is *not* to investigate novel memory-allocation algorithms). `mem_mgr` uses a simple free-list, with very little concern for fragmentation (the only protection is that free-list blocks are not 'split' if the remainder would be less than some minimum size (1kB by default)).

To manage the free-list, `mem_mgr` needs access to all linear memory. Therefore the `mem_mgr` component has a BSS of the maximum amount of linear space, and must be installed at linear address 0. This means that the `mem_mgr` overlaps all other components (including the ORB).

B.3 `scanner`

The `scanner` component is responsible for using code-scanning in order to determine whether or not an untrusted component can safely execute while the microprocessor is in kernel mode (see

Chapter 5).

B.3.1 Overview

This component has two methods. `scan()` is used to scan the code section of a given component image (possibly claiming IO ports), and `release()` releases any IO ports claimed.

Certain components (such as device drivers) require access to certain I/O ports. Access is restricted to specifying the port in immediate form (that is, instructions `in/out reg, imm` are allowed, but `in/out reg, reg` are not). Furthermore, only one untrusted component type may contain instructions accessing a given port at any time. Access is granted on a first-come, first-served basis. If a type is successfully scanned that contains an instruction `in reg,n`, port `n` is *claimed* for read-access. Any subsequent scanning of a component containing a read on port `n` will fail until the component is `released`.

As mentioned in Chapter 5, implementing code-scanning on machines with variable-length instructions is non-trivial. Since IA32 uses variable-length instructions, the code-scanning implemented here uses the method introduced in Section 5.7.2.

B.3.2 Interface

```
interface scanner {
   void ctor();
   void dtor();

   bool scan( objref image, bool trusted );
   void release( objref type );
};
```

```
bool scan( objref image, bool trusted )
```

This method must be called by `mem_mgr` — any attempt to call by another component will trigger an `xcp_gte_prot` exception.

If the component image in `image` contains a code section, and that code section contains any privileged instructions or segment-register loads, `false` is returned. Otherwise `true` is returned.

As well as restricting the use of privileged instructions, `scan` also restricts access to I/O ports. If the `trusted` parameter is `true` any instructions are permitted. However, the type still claims access to I/O ports (even if those ports have already been claimed). Multiple types may claim concurrent access to ports if `scan` is called with `trusted` passed as `true`, but only one type that

calls `scan` with `trusted` of `false` may have access to a given port concurrently, and then only if no privileged components claimed the port previously — that is, regardless of `trusted`'s value when `scan` is called, a 'claim-count' is incremented for each port referenced.

```
void release( objref type )
```

This method must be called by `mem_mgr` — any attempt to call by another component will trigger an `xcp_gte_prot` exception.

This method should be called when `type` is uninstalled if that type has claimed any I/O ports. All ports that are referenced by the type's code section are declaimed.

B.3.3 Implementation

The implementation follows the algorithm outlined in Section 5.7.2.

`scanner` also maintains a 32 bit 'read-claim-count' and a 32 bit 'write-claim-count' for each port using an array of pairs of integers.

B.4 `idisp`

Even the most minimal kernels (such as exokernels [30]) handle interrupts at some level, if only to farm them out to user-level modules. One of the most striking aspects of Go! is that the ORB remains completely oblivious to interrupts. This section describes GTE's `idisp` component, used to handle interrupts.

B.4.1 Overview

On its own, `idisp` will do nothing with interrupts other than to resume execution in the interrupted context immediately after their receipt. In order to perform useful work, components such as device drivers *attach* to the interrupt vector in which they are interested. More specifically, a (blocked) `thread` component is attached that will resume execution in the context of the interested component.

`idisp` is very simple. There is no interrupt priority management: the receipt of an interrupt always results in the attached thread being resumed in a scheduler that does not support thread priorities. There is only limited security policy: interrupts are attached to on a first come, first served basis. That is, once a thread is attached to a given interrupt vector, no other threads may

attach to it until the said thread is unattached. Only the component that attached a thread may unattach it.

B.4.2 Interface

idisp provides methods to attach and unattach thread components and also to indicate that an interrupt has been received.

```
interface idisp {
   void ctor();
   void dtor();

   void attach( thread t, uint vector );
   void detach( uint vector );
   void intr_done( uint vector );
};
```

```
void attach( thread t, uint vector )
```

Thread t will be woken up on receipt of an interrupt on vector. If there is already a thread attached to interrupt vector, an xcp_gte_prot exception is thrown.

```
void detach( uint vector )
```

The thread attached to interrupt vector is unattached. The calling component must be the one that has most recently attached a thread to interrupt vector, otherwise an xcp_gte_prot exception is thrown.

```
void intr_done( uint vector )
```

This method should be called in the context of the handling thread when the interrupt has been serviced. Interrupts on vector are re-enabled, and the thread put to sleep, pending receipt of another interrupt on vector.

If this method is called by a thread other than the one attached to interrupt vector a gte_xcp_prot exception is thrown.

B.4.3 Implementation

In order to understand idisp's implementation details it is first necessary to understand the interrupt mechanism employed by the Intel 80386 and the IBM PC. Such details are explained

here.

The PC interrupt architecture is made up of two parts: the IA32 (CPU) mechanism, and the surrounding hardware defined by the PC (motherboard). Readers familiar with the 8086 interrupt mechanism should note that the interrupt handling mechanism changed radically with the introduction of protected mode on IA32.

The IA32 chip uses vectored interrupts. There are up to 256 interrupt vectors, each with its entry point defined by an entry in the Interrupt Descriptor Table (IDT). The IDT is an array of interrupt or TSS descriptors, and is pointed to by the IDT register (IDTR). On receipt of an interrupt, control is transferred to the code segment and offset specified by the vector's entry in the IDT. The first 32 vectors are reserved by Intel for use with faults.

The IBM PC uses the 8259 Programmable Interrupt Controller (PIC), which manages up to 15 hardware interrupt request lines (IRQs). The IRQs can be redirected to any interrupt vector, and should always map to at least vector 32 to avoid conflicting with Intel's reserved vectors. There are actually two PICs cascaded and multiprocessor motherboards employ an APIC (Advanced Programmable Interrupt Controller), although the details of this are beyond the scope of this overview.

On construction, `idisp` assumes that interrupts are disabled (guaranteed by the ORB). As part of its constructor, `idisp` initialises the 8259 PIC to direct all IRQs to vector 32 and above. `idisp` contains a static array of interrupt descriptors (the IDT) which is initialised so that every interrupt enters the `idisp`, behaving as described below. The data segment's base address is obtained from the `mem_mgr` component and the IDT's linear address calculated. The IDTR is pointed at the IDT linear address, and interrupts are enabled.

On receipt of an interrupt, `idisp` first examines whether or not there is a handler-thread attached to that vector. If not, execution is resumed immediately through issuing an `iret` instruction. If there is a handler-thread registered and the vector is an IRQ (that is, the vector is in the range 32–48), `idisp` programs the PIC to mask out future interrupts on that IRQ. `idisp` then sends the End-Of-Interrupt (EOI) signal to the PIC, indicating that the pending IRQ has been received. `idisp` then calls the `sched` component's `intr` method, passing the ObjRef of the handler-thread. This results in the current thread's suspension and the resumption of the handler-thread. The `intr_done` method is called to indicate that a specific interrupt vector has been serviced, and the handler-thread is ready to receive further interrupts. If `vector` is an IRQ, that IRQ is unmasked on the PIC. Regardless of the vector, control is then `xferd` to `sched`'s

`block()` method.

B.5 thread

The `thread` component is used to keep track of threads, mainly between clients and the scheduler.

B.5.1 Overview

The `thread` interface is designed to be useful for the implementation of more advanced scheduling without modifying `sched`. Unlike `sched`, new implementations of this interface may be freely created by arbitrary components without compromising protection.

B.5.2 Interface

```
interface thread {
   void  ctor( comp start_comp, uint start_method );
   void  dtor();

   void  start();
   void  resume();
   stack get_stack();
};
```

```
void ctor( comp start_comp, uint start_method )
```

The arguments to the constructor are used to specify the method (`start_method`) and component (`start_comp`) in which the thread should start execution when it is first scheduled.

```
void start()
```

`start` is called by the scheduler the first time the thread is executed — if the thread implementation needs to perform any implementation-specific work at this point, this method can be used to perform it. The default implementation does nothing other than calling the method on the component specified during the thread's construction.

```
void resume()
```

Depending on how the thread is attached to the scheduler (see Section B.6), this method might be called before the thread is scheduled. The default implementation does nothing.

`stack get_stack()`

Return the ObjRef of the (first) stack component associated with this thread.

B.5.3 Implementation

The default `thread` is trivial. There are two words of state, used to store the two parameters passed in at construction. This state is referred to during execution of `start()` in order to start the thread in the appropriate component and method.

B.6 sched

This component provides a simple, round-robin, single-priority, preemptive scheduler.

B.6.1 Overview

As described in Section 6.3.4, Go! is implemented so that preemptive multithreading can be added relatively easily. This is implemented in GTE by the `sched` component. `sched` provides preemptive multithreading by attaching to the timer interrupt via `idisp` (Section B.4). `sched` also provides methods to block and unblock threads.

B.6.2 Interface

```
interface sched {
   void ctor();
   void dtor();

   void intr( thread handler );
   void click();
   void attach( thread t, uint flags );
   void detach( thread t );



   void block ();
   void unblock( thread t );

   thread get_current_thread();
};
```

`void intr( thread handler )`

This method should be called in response to receipt of an interrupt to pass control to the thread `handler`.

`intr` must be called with interrupts disabled. Any attempt to call this method with interrupts enabled will result in an `xcp_gte_prot` exception being thrown. This ensures that only privileged components can call `intr()` since only privileged threads will execute with interrupts disabled.

`idisp` calls this method via an `xfer` in response to receipt of an interrupt with a registered handler. The currently executing thread is suspended, and thread `handler` is resumed in its place.

`void click()`

This method is called in response to the timer interrupt. It does nothing except `xfer` control to the `idisp`'s `intr_done` method. Calling this method with a thread other than the scheduler thread *yields* the current time slice (that is, the caller thread is suspended (but left runnable) and another thread chosen).

`void attach( thread t, uint flags )`

Call this method in order to attach a thread to the scheduler. The thread will be started according to the scheduling rules — that is, when its time-slice next arrives; no timing guarantees are made. When its time-slice arrives, thread `t`'s `start` method is called.

The following bits of `flags` are significant:

31 If set, the thread is attached in a runnable state, otherwise the thread is attached blocked.

30 When method `t`'s `start` method returns, thread `t` is deemed to have completed. If bit 30 is set then the thread will be restarted (that is, the thread's `start` method will be called again). The thread starts in its original state — that is, if bit 31 is clear then on completion the thread will be blocked, with a call to `start` pending.

29 If bit 30 is clear then bit 29 defines the action taken on the thread's completion. If set, an `xcp_thread_complete` exception is thrown on completion of the thread. Otherwise thread `t` is destroyed on completion.

28 If set, the thread's `resume()` method is called prior to its resumption at the start of each of thread `t`'s time-slice (with the exception of the first).

27 If set then any other thread can wake the new thread. Otherwise only interrupts can unblock the new thread.

```
void detach( thread t )
```

Call this method in order to detach a thread from the scheduler.

```
void block()
```

Suspends *and blocks* the current thread. The thread will not be run again until it is explicitly woken by a call to `wake`.

```
void wake( thread t )
```

Marks thread `t` as 'runnable'. If there is no thread `t`, or thread `t` is not in a 'blocked' state, an `xcp_gte_invalid` exception is thrown.

```
thread get_current_thread()
```

This method is the equivalent of the ORB's `get_self` — it returns the ObjRef of whatever thread is used to call it.

B.6.3 Implementation

The implementation of `sched` is somewhat intricate, and its interface is intertwined with `idisp`'s[1].

[1] Note that despite the interfaces of `sched` and `idisp` being mutually dependent, the *implementations* are not.

On construction, a 'scheduler-thread' is created, and `sched` attaches this thread to itself so that it will call its `click` method. The 'scheduler-thread' is then attached to the `idisp` on vector 32 — the timer interrupt. This means that the `click` method is called on each timer click (approximately 18.2 times per second). The `click` method does nothing except `xfer` to the `idisp`'s `intr_done` method, which in turn `xfers` to `sched`'s `block` method which chooses a new thread. The result is that each clock results in the currently executing thread being suspended, and a new one is scheduled. Therefore in response to a time interrupt: (1) the current thread is suspended; (2) the scheduler-thread is woken; (3) the scheduler-thread is suspended; (4) the next thread is resumed. This is shown in more detail in Figure B.2.

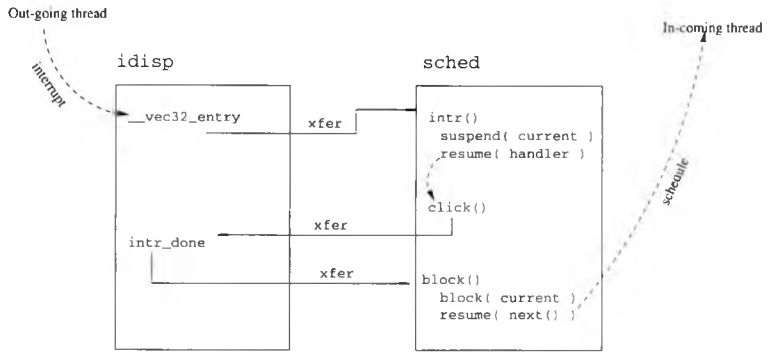


Figure B.2: `sched` and `idisp` interaction

This mechanism results in extra temporal overhead due to the unnecessary resumption of the intermediate 'scheduler-thread' between thread switches. However, this mechanism means that the timer interrupt is handled just as any other interrupt. Furthermore, in a more complete system it is likely that `sched`'s `click` method will perform useful work, such as checking for time-outs.

B.7 `xcp_mgr`

A minimal exception handler. When as exception is caught its details are displayed on the screen and the system halted.

B.7.1 Overview

`xcp_mgr` attaches to interrupt vector 30h via `idisp`. On receipt of an exception, the exception's details are displayed on the screen and the system halted.

B.7.2 Interface

```
interface xcp_mgr {
   void ctor();
   void dtor();

   void handle( objref xcp, uint offs, objref caller, objref callee );
};
```

```
void handle( objref xcp, uint offs, objref caller, objref callee )
```

This method is attached to interrupt vector 30h via `idisp`. On invocation diagnostic information is displayed and the system halted. On entry, `eax` must contain the value 1 (indicating that the method was called via the ORB) — any attempt to call `handle` directly will result in an `xcp_gte_prot` exception being thrown.

B.7.3 Implementation

The implementation of `xcp_mgr` is trivial. The constructor attaches `xcp_mgr`'s `handle` method to interrupt vector 48 via `idisp`. On invocation of `handle` from the ORB the system is halted (interrupts disabled and the `hlt` instruction issued).

B.8 `video`

The `video` component provides a simple text-based console driver for CGA, EGA and VGA colour monitors.

B.8.1 Overview

Every component in GTE makes some use of `video`. The component renders strings, individual ASCII characters, and signed & unsigned integers on the screen. A cursor keeps track of the current output position. Text is scrolled up the screen as necessary.

B.8.2 Interface

```
interface video {
   void ctor();
   void dtor();
```

```
    void print_char( char c );
    void print_string( uint str_len, char str[0] );
    void print_uint( uint i, uint bits );
    void print_int( int i );
    void gotoxy( int x, int y );
};
```

`void print_char( char c )`

The ASCII representation of `c` is placed at the current cursor, and the cursor moved to the next location. If necessary, the screen is scrolled one line upwards. ASCII codes 32 (space) through 126 (~), 8 (backspace) and 10 (newline) are handled correctly — other ASCII codes are not guaranteed to function correctly (although implementations are free to support them).

`void print_string( gstr string )`

The string `string` is taken from stack and displayed on the screen. Each character is printed using `print_char`. The `gstr` type refers to a 32-bit, unsigned integer representing a string's length pushed onto the stack, followed by an ASCII representation of said string.

`void print_uint( uint i, uint bits )`

The unsigned integer `i` is displayed in hexadecimal format, padded to `bits` divided by 4 characters. Each character is printed using `print_char`.

`void print_int( int i )`

The signed integer `i` is displayed using decimal format. No padding is performed (that is, the minimum number of characters needed to represent the number are used). Each character is printed using `print_char`.

`void gotoxy( int x, int y )`

Positions the output cursor at row `x`, column `y`. Subsequent output will start from here.

B.8.3 Implementation

The IBM PC architecture defines that memory from `B8000h` is mapped to the screen for all colour video devices (GTE does not support monochrome video cards). The memory is a row-major, 2d array of 16 bit elements. Even bytes within the array contain the ASCII code of the character

displayed on the screen. Odd bytes contain a set of attributes with which to display the character in the preceding byte. The `video` component implements a `put_char` method that copies ASCII values to the appropriate location within this memory-mapped video space. `put_char` also provides scrolling.

All methods call upon this `put_char` routine to plot characters. `print_int` and `print_uint` convert their parameters to the relevant ASCII codes before passing them to `put_char`.

B.9 keyb

This component provides a simple, interrupt-driven keyboard device driver.

B.9.1 Overview

The component responds to the 'key-down' and 'key-up' interrupts generated by the PC's 8042 keyboard controller, and buffers keystrokes. Arbitrary clients may remove characters from this buffer. Only UK layout keyboards are supported (although it would be trivial to provide slightly altered implementations; one for each layout).

B.9.2 Interface

```
interface keyb {
   void ctor();
   void dtor();

   void reset();
   char getc();
};
```

`char getc()`

This method will return the ASCII code of the first character from the driver's FIFO buffer. If the buffer is empty the thread is blocked, waiting on the arrival of a keystroke. 'Meta-keys' such as shift are not returned, but interpreted appropriately (for example, depressing an alphabetical key while holding shift will produce the capital of that letter).

`void reset()`

Called to reset the keyboard. This routine must be called before the keyboard is guaranteed to function correctly.

B.9.3 Implementation

During construction the keyboard controller is reset, and a keyboard test performed. Assuming all goes well, the keyboard driver's buffer (implemented as a circular array) is initialised. A thread is created and attached to `sched` as an interrupt handler. The interrupt handler thread is then attached to `idisp` on vector 33 (IRQ 1 — the keyboard IRQ).

On receipt of an interrupt, the 'scan-code' is retrieved from the keyboard (using IA32's I/O ports), and examined. If either shift key is depressed, the 'shift-flag' is set. When the shift key is released, the 'shift-flag' is cleared. If the scan-code denotes an alphabetic key, it is converted to the appropriate ASCII code (capital thereof if the 'shift' flag is set), and stored in the buffer. Appropriate conversions are also applied to the number and punctuation keys if the shift flag is set. The condition variable associated with the buffer is also set in order to wake any threads waiting for keyboard input.

B.10 `cli`

`cli` is GTE's Command Line Interface. It obeys just 5 commands: `help`, `stress`, `tt`, `exit` and `reboot`.

B.10.1 Overview

`cli` simply takes keystrokes from the `keyb` component and interprets them. If the user types `help`, the following message is displayed:

```
GTE Command Line Interface.

The following commands are recognised:
  help   - print this message
  stress - perform a stress-test
  tt     - timing test: report on times taken by various operations
  exit   - quit GTE
  reboot - reboot the system
```

If any command is entered that is not recognised, the `cli` displays the message:

```
Command not recognised
```

The 'stress-test' performed is described in Section 7.4. The 'timing-test' measures the time taken (in cycles) for various operations. The tests and their results for the latest version of GTE are documented in Chapter 7.

B.10.2 Interface

```
interrface cli {
   void ctor();
   void dtor();

   void process();
};
```

```
void process()
```

This method repeatedly reads and processes lines of input from the keyboard. If the user types `reboot` the system is rebooted. If the user types `help`, the help message is displayed. If the user types `exit` then the method returns. Typing `stress` or `tt` results in the execution of a stress-test or the timing tests respectively. If anything else is typed, the error message is displayed. The routine repeats until the `exit` command is typed.

B.10.3 Implementation

The `cli` implementation is trivial. The `process` method simply takes characters from `keyb`'s `getc` method, searching for valid commands when a new-line is received.

Appendix C

Object Orientation on Go!

C.1 Introduction

Over the last decade, object orientation has become the software-engineering tool of choice. It is therefore desirable that the Go! component-model is (or at least can be) viewed as object oriented by the programmer. However, object-orientated programming should not be mandated, and should not adversely effect performance, at least if not used. Therefore, there are two aspects to Go!'s component model: *the programmer's view*, and *the implementation.*

High-level programming abstractions like inheritance are not supported directly Go! Instead, the ORB implements only rudimentary support for object-orientation, on top of which the programming environment (specifically the "IDL-compiler"[1]) can create the illusion of a fully-featured, object-oriented system. Providing only *support* for object orientation (rather than full object orientation) at the ORB level improves both performance and flexibility. Performance because the ORB is not weighed down by expensive and complicated features such as multiple-inheritance, and flexibility because the exact component-model used can be specified by the programmer. For example, two programmers might access the same service differently: one using an object-oriented IDL compiler that generates C++ stubs, and one using a procedural IDL compiler that generates C stubs.

This appendix is divided into two parts: *the programmer's view* which specifies the features that programmers can use, and *the implementation* which explains how the "IDL compiler" can

[1] An IDL compiler is a tool that takes as its input a specification of a component's interface in some 'Interface Definition Language' (IDL) and outputs 'skeleton code' and 'header' files which abstract the implementation to aid the programmer significantly.

use this to present the *the programmer's view* on *the implementation.*

C.2 The Programmer's View

The component-model presented to programmers is similar to that found in most object-oriented systems. Each component is an instance of an *implementation* and each implementation realises an *interface.* An interface is a set of method signatures and is defined in some Interface Definition Language (IDL). The interface defines a *contract*, which is a semi-formal[2] definition of the behaviour of the component and its methods. Both interfaces and implementations may inherit singly or multiply.

C.2.1 Interfaces and Implementations

Each component has an *interface* and an *implementation* (in fact, a component may have several of each, as described in Sections C.2.3 and C.2.4). The interface's methods (functions) are realised by the component's implementation. The interface is properly separated from the implementation (that is, unlike models such as C++, there is no way for other components to access the implementation's state other than by calling methods on the interface).

Each interface defines 2 'special' methods: the *constructor* and *destructor.* The constructor is the first method to be called after the component instance is created, and the destructor the last method to be called before the instance destroyed. Methods other than the constructor and destructor are called by other component implementations (via the ORB).

C.2.2 Interface Inheritance

Interface inheritance allows a new interface to be derived from an existing one (that is, standard inheritance as found in most object-oriented systems). The new interface contains all of its base's methods, and optionally adds some of its own. The derived type has is a *sub-type* of its base. In other words, the derived type has an *is-a* relationship with its base — if B derives from A, then B *is-a* A. This means that anywhere a component of type A is expected, one of type B may be used instead. More formally, any use of B requires no additional pre-conditions than the corresponding use of A, but pre-conditions may be dropped, and post-conditions may not be loosened on B, but they may be tightened.

[2]The IDL specifies the syntax formally, but the semantics informally.

Interfaces may inherit indirectly (that is, C may inherit from B which inherits from A) and multiply (that is, interface E can inherit from D *and* C).

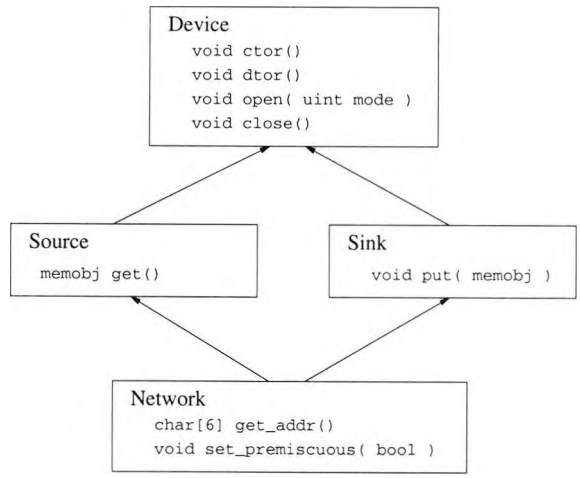


Figure C.1: An example of indirect and multiple interface inheritance

As an example, take the inheritance graph as shown in Figure C.1. The network device driver can be used anywhere a data sink or source can be. As with all devices in this example, it is necessary to open and close the network device before and after use. Note that if several interfaces are inherited from that themselves inherit from the same interface (as shown in the 'diamond' pattern of Figure C.1), there is only one instance of the base interface (known as *virtual inheritance* in C++). If several instances of one interface are required, the object can export multiple interfaces (see Section C.2.4).

C.2.3 Implementation Inheritance

It is sometimes necessary for an implementation of a derived interface to realise its base methods in terms of an existing base implementation. Figure C.2 gives an example of this for a graphical user interface (GUI), where the interface `appl_wnd` is used to realise application-specific windows (the inset gives the IDL of these components). In this example, the `appl_wnd` interface derives from `window` which defines the standard windowing behaviour. The base interface is extended with a `paint` method that is called whenever the application should re-paint the contents of its main window (note that this differs from the base `draw` method that is used to draw the window border and controls). It is undesirable for each implementation of `appl_wnd` to be required to implement the entire windowing behaviour — instead such responsibilities should be delegated to a default `window` implementation provided by the GUI. The example shown in figure C.2 (a) demonstrates

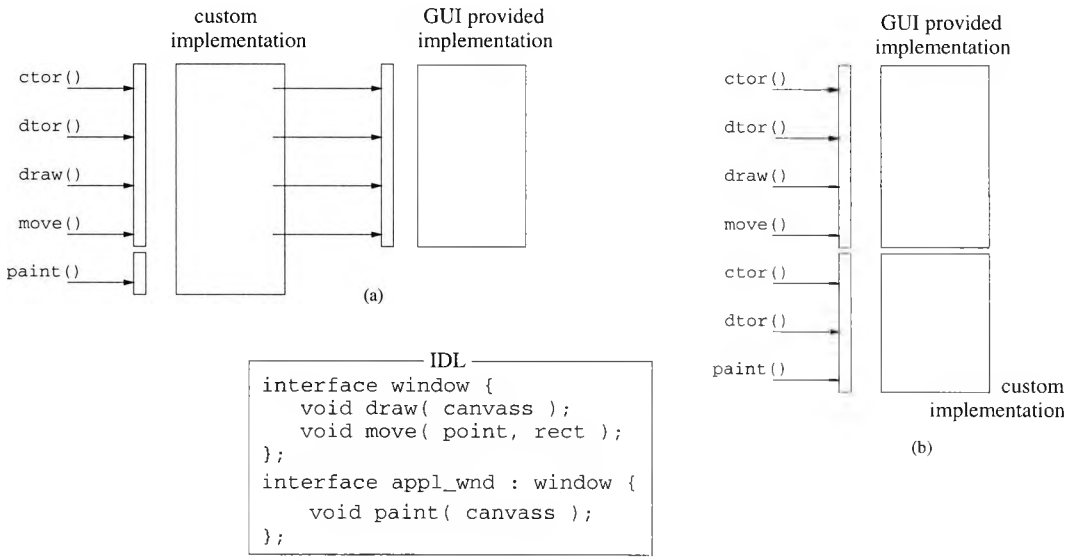


Figure C.2: Implementation inheritance vs delegation

the case when an implementation of a derived interface delegates the base behaviour to an instance of the base implementation. This requires two extra context switches per base method call (one each for the call to, and return from the GUI provided `window` implementation). Although a single context-switch in Go! is relatively cheap, these extra context switches can become significant when there are several levels of inheritance with delegated behaviour.

The example in figure C.2 (b) demonstrates implementation inheritance. Here, the base methods point directly to the GUI-provided implementation. This means that calling base methods (those implemented by the GUI library) requires no extra context switches. Note that the GUI and application supplied implementations are still protected from each other. Note also that each inherited implementation requires its own constructor and destructor methods.

C.2.4 Multiple Interfaces

Just as implementation inheritance results in a component with multiple implementations, a component can support multiple interfaces. More specifically, several interfaces can point to the same implementation, and one interface can point to several implementations: interfaces map to implementations in a many to many fashion. Multiple interfaces are useful in cases where multiple views onto the same object are required. As an example, take a proxy implementation of a component on a remote machine. The proxy component provides 'location transparency' — clients of the component are unaware that it is resident on a remote machine. The proxy realises the interface

by directing requests over the network to the 'real' implementation on another machine. Such a proxy implementation might also have methods to interrogate the physical location of the remote component, as illustrated in Figure C.3. As another example, a 'file' component might provide a `POSIX` and a `native` interface.

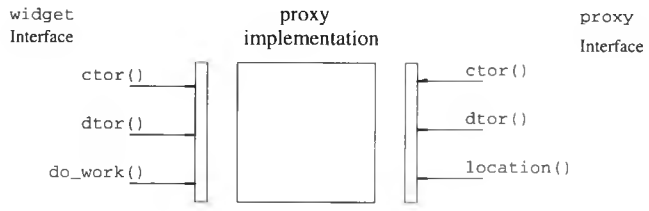


Figure C.3: Multiple Interfaces

Multiple interfaces are also useful when evolving an interface during hot-swapping. The incoming implementation can export a new (modified) interface as well as the original one to continue supporting legacy clients.

C.3 Object Model Implementation

The object model described in Section C.2 is a conceptual model that is supported at the programming level if required. Go! does not support object orientation directly partly because this would compromise efficiency, but also because object-orientation is closely linked to type-safety, and so is better implemented at the language level.

However, Go! does provide support so that the above object model can be implemented easily. This is achieved by allowing several components to be created atomically, all with consecutive references. A separate interface is created for each possible implementation inherited. If implementation inheritance is employed, each interface references a different implementation. Otherwise, each interface refers to the *same* implementation. For the GUI example in Figure C.2, Go! would be called upon to create the GUI implemented `window` component and the application realised `appl_wnd` component, with consecutive references. If implementation inheritance is not required, and `appl_wnd` is required to implement all methods, and both the `window` and `appl_wnd` interfaces reference the same implementation. An automatically generated C++ header file for this case might look like:

```
class window {
public:
    void draw( canvass &c )
```

```
            { push( c ); orb::call( (objref)this, 0 ); }
    void move( point p, rect r )
            { push( p.x, p.y, r.x, r.y ); orb::call( (objef)this, 1 ); }
};

class appl_wnd : public window {
public:
    void paint( canvass &c )
            { push( c ); orb::call( ((objref)this) + 1, 0 ); };
};
```

Note that the class's `this` pointer is not really a pointer, but the `window`'s ObjRef. Because `appl_wnd` is one level down inheritance hierarchy, its methods are accessed via the object's second interface, hence 1 must be added to `window`'s ObjRef when calling the `appl_wnd`'s methods. Note also that although the example is given in C++, the mapping of object-orientation to components is not language specific: that is, other object-oriented languages could be used (although the details would differ).

Note also that because linear space is not managed by the ORB (see Section 6.3.4) no explicit support is necessary for allowing multiple interfaces onto one implementation. In order to create two references both pointing at the same implementation, both references' data and code segments should overlap. It is left to whatever component is responsible for memory management to orchestrate such overlapping segments.

C.3.1 Multiple Inheritance and Polymorphism

The above scheme needs to be extended to cope with multiple inheritance. This is because a given interface's reference is calculated from the sum of its base interface's reference, and the level in the inheritance hierarchy of the given interface. However, with multiple inheritance this means that distinct interfaces are required to share a reference.

Figure C.4 shows an example of this. Because `sink` and `source` are both at level 1 in the hierarchy, they share a reference. To overcome this, the base reference is duplicated, as shown in Figure C.5. Note that although there are two references to the `sink` interface (103 and 105), these refer to the same implementation (that is, references 103 and 105 are two (identical) interfaces onto one implementation).

Polymorphism means that a client must be able to use a specialisation of a more general type without being aware. In other words, wherever an object of some type is used, a specialisation of that type (that is, a derived interface) can be used transparently. However, clients must be careful

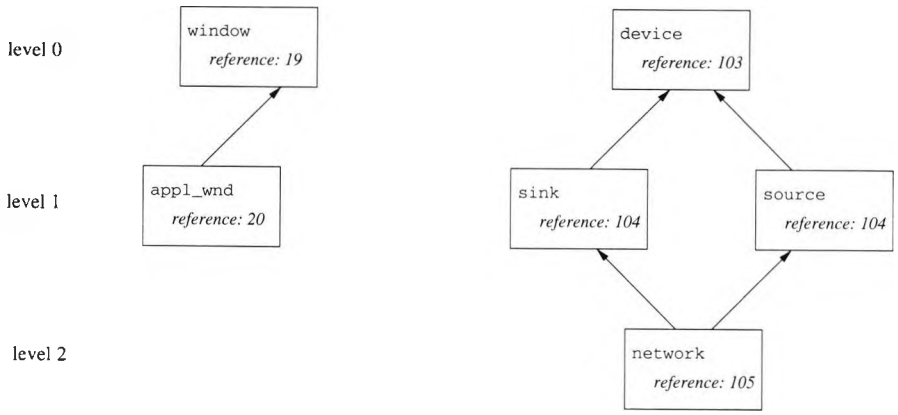


Figure C.4: Multiple inheritance with several interfaces at one level

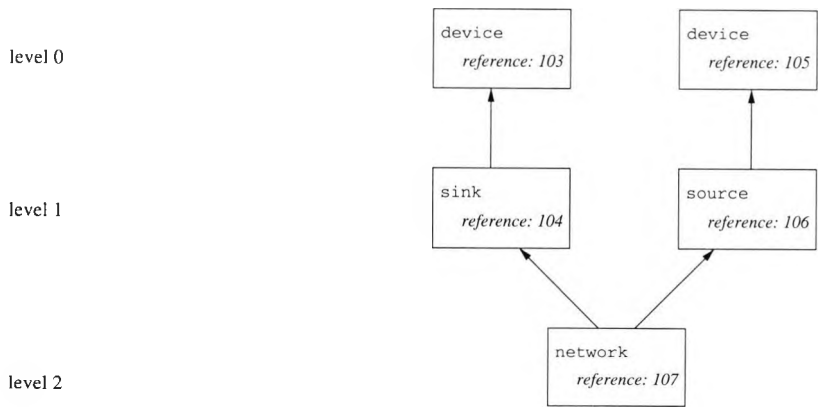


Figure C.5: Duplicating interfaces in multiple inheritance

to *cast* the specialised type appropriately. For example, imagine two methods `foo` and `bar` that each take 1 parameter of the type `source` and `sink` respectively. Polymorphism dictates that a component of type `net` can be supplied instead to either, but in doing so, one must be careful to modify the reference appropriately. Specifically, with the example in Figure C.5, `foo` must be passed reference 105, and `bar` reference 103.

Fortunately, object-oriented compilers are required to exhibit this behaviour too. For example, the following C++ will be compiled correctly:

```
/*
 *IDL compiler generated code
 */
class device {
    char dummy;
public:
    void open( int mode )
```

```
        { orb::call( (objref)this, 0, mode ); };
    void close()
        { orb::call( (objref)this, 1 ); };
};

class sink : public device {
    char dummy;
public:
    void   put( memobj m )
           { orb::call( (objref)this + 1, 0, m ); };
};

class source : public device {
    char dummy;
public:
    memobj get()
           { return orb::call( (objref)this + 1, 0 ); };
};

class net : public sink, public source {
    char dummy;
public:
    char[6] get_addr()
            { return ret_obj<char[6]> orb::call( (objref)this + 2, 0 ); };
    void    set_premiscuous()
            { orb::call( (objref)this + 2, 1 ); };
};

/*
 *Note: the following code is not automatically generated, but is here to
 *demonstrate the compiler's behaviour with the above code.
 */
void write( sink &s );
void read( source &s );
void connect( net &n );


void
main() {
    net &n = *(net*)103;
    write( n );             //will be passed ref 103
    read( n );              //will be passed ref 105
    connect( n );           //will be passed ref 103
}
```

Note that each interface has a `char` member, `dummy`. This forces the compiler to offset the object's `this` pointer by the appropriate amount in order to obtain the correct ObjRef for the interface's level in the inheritance hierarchy. Care must be taken to ensure that the compiler does not 'pad' the offsets, as C and C++ compilers are free to do (the above example is known to work correctly with `gcc`).

C.4 Summary

This appendix has presented how object-oriented compilers can be used to provide programmers an object-oriented view of Go! while only minimal support is provided by the ORB (namely the ability to allocate consecutive references at component instantiation). Providing rudimentary support like this rather than full-blown support increases both performance of the ORB, and flexibility of the programmer's view.