ADDING OPERATING SYSTEM STRUCTURE TO LANGUAGE-BASED PROTECTION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Christopher Kirk Hawblitzel

August 2000

ADDING OPERATING SYSTEM STRUCTURE TO LANGUAGE-BASED PROTECTION

Christopher Kirk Hawblitzel, Ph.D.

Cornell University 2000

Extensible Internet applications increasingly rely on language safety for protection, rather than using protection mechanisms based on virtual memory. For example, the Java language now provides protection for applets, servlets, agents and active networks. Unfortunately, typical safe language systems do not have the support for resource control, revocation, and on-demand program termination present in traditional operating systems. Naive attempts to add these features to a language can easily backfire: Sun recently deprecated Java's thread termination methods because of subtle interactions with Java's synchronization and abstraction mechanisms.

In this thesis, I argue that these problems arise from a lack of clear structure in safe language systems. Often, these systems allow objects, threads, and code to pass freely from one program to another, blurring the boundaries between programs. To restore the boundaries, I introduce the idea of a safe language *task* that encapsulates a program's objects, threads, and code, and I argue that a system based on the task model can provide strong and simple guarantees about resource control, thread management, termination, revocation, and whole-program optimization. I present two implementations of the task model: the J-Kernel, which uses Java remote method invocation for inter-task communication, and Luna, which extends Java's type system to express the task model directly.

BIOGRAPHICAL SKETCH

Chris Hawblitzel was born in Kansas City, Missouri in 1973. He started college at UC-Berkeley in 1989 and received a bachelor's degree in physics there in 1993. He began graduate school at Cornell studying computational physics, and switched to computer science in Spring of 1996.

ACKNOWLEDGMENTS

There are so many people to whom I owe thanks that I cannot hope to repay them all here. Most of all I would like to thank my parents, who introduced me to computers in the first place, for their boundless support and encouragement, their warm sense of humor, and their unfailing integrity. My mother and father are lifelong teachers and lifelong learners—it was they who taught me how to learn and from them I learned how to teach.

Many thanks go to Thorsten von Eicken and Greg Morrisett for opening up entire new worlds to explore, and for straightening every crooked path that I traveled on. Their ideas formed the basis for this thesis, and their advice molded my own shaky proposals into something solid and practical. Working with them was both an honor and a pleasure.

I am greatly indebted to Greg Czajkowski, Chi-Chao Chang, Deyu Hu, and Dan Spoonhower for transforming the J-Kernel from an isolated prototype into a real working environment. Without them, the J-Kernel would never have hatched from its shell.

Thanks to Lidong Zhou, Ulfar Erlingsson, Fred Schneider, Andrew Myers, Brian Smith, Hal Perkins, Jay Lepreau, Wilson Hsieh, and Pat Tullmann, whose insights and comments significantly influenced the content of this thesis.

Thanks to the Marmot group for sharing a stellar compiler with us.

Thanks for moral and practical support to everyone in Greg Morrisett's group, including Fred Smith, Dave Walker, Neal Glew, Dan Grossman, Stephanie Weirich, Steve Zdancewic, and Karl Crary, as our own group was slowly decimated by graduation and employment. Thanks to Greg

Czajkowski, Chi-Chao Chang, Deyu Hu, Soam Acharya, Sugata Mukhopadhyay, Wei Tsang Ooi, Eva Luther, and Dan Spoonhower for such an intellectually stimulating and enjoyable work environment, particularly when the work consisted of discussing weighty topics like the future of electronic commerce or the virtues of Polish Hip-hop. Finally, thanks to my officemates, Tibor Janosi and Matt Fleming; I should probably drop by someday. Matt, of course, knows what's in the black bag, but others can savor the mystery.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE:
INTRODUCTION

Modern programming languages are often expected to perform traditional operating system duties, but they are not ready to replace operating systems yet. This thesis argues that programming languages must incorporate additional operating systems features if they are to implement operating system functionality.

The distinction between programming languages and operating systems used to be clear. Programming languages were tools to write applications without having to enter assembly code by hand, while operating systems were tools to manage a system's resources and protect one application from another. These days, however, developers of extensible applications often expect programming languages to perform functions traditionally associated with operating systems. Browsers, servers, agent systems, databases, and active networks all support extensions written in a safe language such as Java. In these systems, the language's type safety protects the application core from the extensions (and the extensions from each other) by restricting the operations that extensions are allowed to perform.

However, type safety does not prevent a program from consuming too many resources, or ensure that runaway program execution can be stopped cleanly. Nor does type safety provide a large-scale framework for structuring communication between extensions. In addition to type safety, language-based protection systems need an aggregate structure that encapsulates a

program's resources, and guides program termination, resource management, security analysis, and communication between programs.

Existing aggregate structures, such as Java's *class loaders* and *thread groups*, are ad hoc and suffer from undesirable and unexpected interactions with the language's type system [Sar97, Javb]. The main weakness of these existing structures is that they manage subsets of a program's resources in isolation: class loaders track a program's code, but say nothing about a program's threads and objects, while thread groups track a program's threads but not its code or objects. Thus, current structures cannot make guarantees about the interactions between code, threads, and objects. This thesis makes two contributions toward solving this problem:

- It proposes a more comprehensive aggregate structure, called a *task*, which controls a program's code, objects, and threads together. In this respect, a task is analogous to a *process* or *task* in a traditional operating system.
- It proposes inter-task communication mechanisms that distinguish code, objects, and threads belonging to one task from those belonging to other tasks. This clarifies the boundaries between tasks and allows the system to guarantee properties of tasks. For example, the systems presented in this thesis ensure that a task's threads only execute the task's own code. Unlike current Java communication mechanisms, the communication mechanisms presented in this system are general purpose: they may be used as easily for agent-to-agent communication as for applet-to-browser or servlet-to-server communication.

The rest of the thesis is as follows: chapter 1 motivates and describes language-based protection and the task model in detail. Chapters 2, 3, and 4 describe two particular implementations of the task model, the J-Kernel and Luna. Chapter 5 explores the task model in more generality, discussing related work and alternate approaches to implementing the task model. Chapter 6 concludes.

1.1 Motivation: the challenges of a programmable internet

Operating systems with complex protection mechanisms, such as Multics [CV65], were designed for large, centralized, multi-user compute servers, where the OS had to prevent one errant program from disrupting other users. The past 20 years have seen a shift away from centralized computation to networks of single-user personal computers. PC's often run unprotected operating systems like Windows 98 or MacOS, partially because networking protocols allow individual computers to fail without disrupting the whole network. Mass acceptance of fully protected PC operating systems, such as NT and Linux, has been slow relative to the rapid acceptance of the PC's themselves. The trend towards small, distributed computers continues with the spread of hand-held and embedded processors. In light of these developments, why worry about protection at all?

The continuing need for protection in a network is due to the spread of executable content and mobile code. First, several forms of executable content are already in wide use: web pages commonly contain Javascript code or Java applets, e-mail often contains executable attachments, and word processor documents often contain postscript code or macros. Such content has already caused serious network-wide damage. In particular, e-mail attachments are

often executed with no protection at all, and are therefore vulnerable to malicious viruses. Second, the Internet has shifted functionality from users' desktops to remote web sites, which limits a user's ability to customize applications (major web sites such as Yahoo, Altavista, Amazon, and Hotmail often contain large data sets, vigorously maintained availability, and proprietary functionality, which cannot be practically transferred to user PC's). Many people have advocated mobile code, such as agents [GKC+98], servlets [Java], active database extensions [GMS+98], and active network extensions [Wet99], to restore a user's ability to customize applications.

In order to execute code embedded in a document, a user must choose to download a document and execute the code. This makes attacks more difficult; a hacker's only hope is to post or send a document with embedded code, and then hope that someone will choose to execute the code.

Agents, servlets, active database extensions, and active networks take the opposite approach, where an extension is actively injected into a remote host, and begins execution with no user intervention. This is inherently more susceptible to attacks, especially in systems that allow anyone to upload code. Even if such systems require authentication before allowing code to execute and perform auditing to track and deter malicious users, protection is important, because hackers often find ways to circumvent authentication and auditing mechanisms. Typical attacks rely on the fact that networking software often has bugs, is misconfigured, or protected by weak passwords [Spa89]. By increasing the programmability of the Internet, active extensions add another weapon to a hacker's arsenal. For example, suppose that Zeke wants to attack Kim's database, and that Kim's friend Alex is trusted to load Java extensions to Kim's system. If Zeke can guess Alex's password, then he

can load an extension onto Kim's system, which can then perform a denial of service attack. The protection offered by Java in Kim's system is still essential to prevent Zeke from attacking the database's integrity and secrecy, although it is not sufficient to protect the database's availability. By contrast, if Kim allowed friends to upload unsafe code to her system, then her system's integrity and secrecy would be violated as well—in effect, her security would become as weak as the weakest of her trusted friends (which in turn are as weak as the weakest of their trusted friends). In addition, even trusted code may contain bugs; protection mechanisms limit the disruption caused by a trusted, but faulty, extension.

1.2 Protection mechanisms

The previous section motivated protection in extensible internet applications. This section looks at several different potential protection mechanisms for such applications: virtual memory protection, software fault isolation, capability mechanisms, and language-based protection.

	hardware enforcement	software enforcement
address-based	virtual memory	SFI
type-based	capability systems	language-based

Figure 1.1: Classifying protection mechanisms

Broadly, these mechanisms are classified into "address-based" protection and "type-based" protection on one axis, and hardware enforced and software enforced on another axis, as illustrated in Figure 1.1.

1.2.1 Address-based protection mechanisms

Current operating systems rely on protection mechanisms provided in hardware, specifically, memory protection, user/supervisor execution levels, and traps to switch levels. Memory protection is based on checking the validity of addresses issued by every instruction accessing memory. The user/supervisor privilege levels provide one execution mode where the access privileges are enforced and one where they can be changed. The traps (or call gates) provide a controlled transfer from user to supervisor level.

The key aspect of address-based protection is that it is based on the values of addresses issued by a program—these are checked and possibly translated—and not on how the program uses the data stored at those addresses.

1.2.1.1 Virtual memory protection

In current microprocessors, the address-based protection is implemented as part of the virtual memory hardware. In CISC microprocessors, the set of accessible addresses (i.e., the address space) is represented by the page tables, while in RISC microprocessors with a software-managed TLB it is represented by the current set of valid TLB entries. The common characteristic of all these implementations is that the granularity of the protection mechanisms is a virtual memory page frame on the order of a few kilobytes, 4KB being the most typical. While it is possible to use smaller page frames in some implementations, this is generally inefficient.

1.2.1.2 Software fault isolation

Address-based protection can also be implemented in software using a technique called software fault isolation (SFI) [WLA+93, Sma97]. In SFI, an executable is analyzed and modified to enforce that all memory references lie within the address space. Part of the enforcement is by analysis and part is by the insertion of explicit address checking or "sandboxing" instructions. These extra instructions ensure that the value of an address in a register lies within the bounds held in other registers that are protected from program modification. A privileged execution level is implemented by instruction sequences that are inserted without being subject to SFI. The details of these instruction sequences depend highly on the characteristics of the machine language being manipulated. For example, SFI is much more efficient on RISC processors with fixed sized instructions than on CISC processors with variable sized instructions. Using SFI, the granularity of address space is generally larger than with hardware-based mechanisms (e.g., an SFI address space is typically larger than a hardware page) because the cost of enforcement increases with the number of regions.

1.2.2 Type-based protection mechanisms

Address-based mechanisms perform access control on the addresses that a program references, without worrying about how the program generates addresses or what a program stores in the addresses to which it has access. By contrast, a type-based protection mechanism limits a program's ability to create values, by providing only a limited set of operations to manipulate values. For example, two integers may be added together to create a new integer, but two addresses cannot be added together to create a

new address. This section considers two classes of type-based protection: capability systems, which rely largely on hardware mechanisms to enforce the type system, and language-based protection, which relies mostly on software enforcement.

1.2.2.1 Capability systems

In a capability system [EB79, Lev84, PCH+82, Red74, WLH81], data values are distinguished from pointers (capabilities) and the system enforces that data values cannot be used as capabilities. The basic idea in such systems is that a program must first acquire a *capability* to a resource in order to access the resource. The type system ensures that a program cannot forge a capability to a resource that it should not have access to.

Some capability systems add a tag bit to every word in memory to distinguish data and capabilities [CKD94], and the processor restricts the operations allowed on values that are tagged as capabilities. In practice, however, this has led to complex processor architectures that are tied to particular protection models or programming languages [PCH+82], and expensive custom memory technologies.

Other capability systems use an ordinary processor's virtual-memory features to partition memory into data and capability segments. While this led to performance problems in early systems [WLH81], because a program must trap into the kernel to manipulate capabilities, more recent systems [SSF99] have argued that advances in microkernel design make the cost of such traps acceptable. A more difficult problem, though, is that segregated data and capability segments complicate the programming model, because capabilities cannot be easily mixed with data in data structures. One possible

solution to this problem is to use cryptographic capabilities [HEV+98, TMvR86], which are large random numbers that are nearly unforgeable because they are difficult to guess. Such capabilities are easier to store in data structures, since they are just numbers.

1.2.2.2 Language-based protection mechanisms

Language-based protection relies on the safety of a programming language's type system to restrict the operations that a program is allowed to perform. The language provides a set of types (integers, functions, and records, for instance), and operations that can manipulate instances of different types. Some operations make sense for some types but not others. For instance, a Java program can invoke a method of an object, but it cannot perform a method invocation on an integer.

Type-safe languages implement capability based access control very naturally: a pointer (also called a *reference* in Java jargon) cannot be forged, and can therefore serve as a capability. Languages typically provide additional access control mechanisms, such as Java's `private`, default, `protected`, `public`, and `final` access qualifiers that specify which code has access to which fields and methods of an object. Wallach et al [WBD+97] discusses Java access control mechanisms in detail. One can design safe languages that have more sophisticated access control. For instance, Jones and Liskov [JL78] extended a language's type system to talk about access rights directly, to provide a similar functionality to advanced capability systems such as Hydra [WLH81]. Recently Myers and Liskov [ML97] have extended this idea to cover information flow control.

A key issue of language-based protection is ensuring that type safety and access control are enforced. Type enforcement may be done dynamically or statically. In the former case, a compiler inserts run-time checks into the compiled code. In the latter case, the compiler analyzes the program to ensure that it respects the type system's constraints without requiring explicit run-time checks. Static type enforcement is a key advantage of language-based protection over traditional capability systems, whose dynamic checks are difficult to implement efficiently without special hardware.

In practice, most programming languages use a combination of static and dynamic enforcement. For instance, Java can statically verify that a variable of type `String` will always point to a `String` object or contain the null pointer. On the other hand, typical Java implementations perform dynamic bounds checks when an array is accessed. Some languages, such as Scheme, perform almost all type enforcement at run-time, which tends to slow program execution.

Safe languages hold many potential advantages over other protection mechanisms. Since all programs run in a single address space, programs can communicate without expensive address space switches. While IPC (inter-process communication) is often an expensive operation in virtual memory based systems, in a safe language system a call from one program to another may be a simple function call. Furthermore, data can be shared between programs at a fine-grained level; programs can share individual objects rather than entire virtual memory pages. For example, ethernet cards typically receive network packets smaller than a page size, which means that traditional operating systems must either copy each incoming packet into application memory to prevent one application from seeing other applications' packets

that landed on the same page, or waste space by receiving only one packet on each page.

Modern programming languages are good at both expressing and enforcing abstractions. Programs aren't limited to sharing arrays of raw bytes: the language can statically enforce the integrity of shared *abstract data types*, which allows programs to communicate at a higher semantic level. For example, the IOLite [PDZ99] and FBuf [DP93] operating system facilities are based on linked lists of buffers, and the validity of the linked lists must be checked (e.g. for non-circularity) when the lists are copied from one process to another. A safe language can statically enforce invariants such as non-circularity of a list.

Safe languages also make it easier to write more flexible, secure programs. They guard against programming mistakes that lead to vulnerabilities. For example, guaranteed array bounds checks guard against buffer overflows that have been exploited by attackers in traditional systems [Spa89]. Typical safe languages support advanced features like object-orientation and higher order functions, which aid the construction of extensible systems, and naturally accommodate dynamic loading.

Since safe languages hide the details of the underlying processor, programs written in these languages tend to be portable across a wide variety of architectures, including small, embedded systems that lack the hardware and OS infrastructure needed for hardware-enforced protection. This portability is an advantage for mobile code systems.

1.3 Where does language-based protection fall short?

The previous section argued the advantages of language-based protection over other protection mechanisms, but unfortunately, language-based protection has serious drawbacks. The performance of safe languages tends to lag behind lower level languages like C, typical language-based systems support only one language, and relying on a language for protection requires trusting that the language's compiler (the just-in-time compiler in Java's case) and run-time system are written correctly. Other research has made great strides in attacking these problems ([BSP+95], [HLP98], [MWC+98], [NL98], [Sha97], [WA99]). This thesis focuses on a different set of problems, arising from the lack of OS features and structures in current language-based systems. Current language-based systems do not adequately support termination, revocation, resource accounting, optimization in the presence of dynamic loading, and analyzing the aggregate system structure.

1.3.1 Termination

Java's mechanism for terminating programs is based on *thread groups*. For example, a web browser creates a thread group for each applet that it downloads. As the applets execute, Java's run-time system places each thread spawned by an applet in the applet's thread group. To shut down an applet, the browser terminates the applet's thread group, which stops all of the applet's threads. While this approach to termination seems reasonable on first glance, closer inspection reveals several difficult problems:

- The wrong code gets interrupted: in Java, a call from an applet to the browser is just a function call, so that the browser code runs in the applet's thread. The applet can kill or suspend the

thread while the browser code is running, possibly leaving the browser in an inconsistent or deadlocked state.

- Malicious code eludes termination: terminating a malicious program's threads does not terminate the program's code, which may still exist in objects that the malicious program created and passed to other programs. If one of these other programs invokes the rogue object methods, the malicious code is revived. Java mitigates this problem by making key datatypes final (e.g. `String`, array types) or primitive (e.g. `int`, `float`) so that they cannot contain overridden methods, but a purer object-oriented language would have more difficulties.
- Upcalls and peer-to-peer communication are not supported: a browser cannot use one of its own threads to call an applet, because this would give the applet the ability to execute in a thread outside the applet's thread group, and this execution would not be stopped when the applet's thread group is terminated. Instead, the browser must transfer control to an applet thread before calling any applet code, which is cumbersome, slow, and requires the programmer to carefully track which method invocations might call applet code. Mutually suspicious peer-to-peer communication (e.g. applet-to-applet or agent-to-agent communication) is even harder, because neither side trusts the other side to perform the needed thread switch.
- Damaged objects violate abstract datatype integrity: under Java's synchronization mechanisms, a thread may enter an

object's method, acquire a lock, start an atomic operation, and then get terminated in the middle of the operation, releasing the lock and leaving the object in a "damaged" state that violates its intended abstraction.

- Resources are not reclaimed: when a traditional operating system shuts down a process, both the process's threads and memory are shut down. Java's termination is weaker, stopping threads but not necessarily reclaiming other resources. Java's `String.intern` method is an example of this: any program can use this method to add strings to a system-wide hash table, and these strings are not garbage collected even when the program exits. Aggressive resource reclamation is more difficult in a language-based environment, which encourages sharing data between programs, than in a traditional operating system based on explicit IPC (although newer, single address space operating systems also encourage sharing—see Mungi [HEV+98], for example).

Because of the problems with thread groups, Sun recently deprecated the `stop` method in the classes `Thread` and `ThreadGroup`, leaving no officially sanctioned way to stop an applet [Javb].

1.3.2 Revocation

In ordinary Java, pointers can serve as capabilities, but once a program is given a pointer to an object, that pointer cannot be revoked. Revocation is a desirable feature for several reasons. First, it helps to implement the principle of least privilege, since it is better to give someone access to something for

only the necessary duration and then revoke the access than to give them access forever. Revocation also accommodates changing preferences over time. An agent once considered trustworthy may abuse the trust and necessitate revoking its privileges. At a lower level, revocation is a good way to give someone temporary, fast access to a resource, such as idle network buffer space or a rectangle in video memory. In addition to these device-specific examples, revocation is also used in general purpose operating system mechanisms: for instance, FBufs [DP93] dynamically revoke write access to buffers when data is transferred between protection domains. Finally, revocation is necessary for termination: access to any services exported by a program must be revoked when the program is stopped, because these services are no longer available after the program's code is unloaded.

Revocation has historically been a problem for capability systems, and it is particularly difficult at the language level, because every pointer could potentially be used as a capability. While adding a level of indirection to every capability may be a way to implement revocation in an operating system with coarse-grained capabilities [Red74], adding a level of indirection to every Java pointer is undesirable.

Naturally, the programmer can add a level of indirection only to those objects whose protection is critical. Figure 1.2 shows how a revocable version of the earlier class `A` can be created. Each object of `A` is wrapped with an object of `AWrapper`, which permits access to the wrapped object only until the `revoked` flag is set. However, it is not always obvious which objects require such protection. Vitek et al [BV99] argue that critical, but unprotected subobjects often leak out of a protected object through field references and method invocations.

```
class A {
  public int meth1(int a1, int a2) {…}
}

class AWrapper {
  private A a;
  private boolean revoked;
  public int meth1(int a1, int a2) {
    if(!revoked) return a.meth1(a1, a2);
    else throw new RevokedException();
  }
  public void revoke() { revoked=true; }
  public AWrapper(A realA) {
    a = realA; revoked = false; }
}
```

Figure 1.2: Using indirection to implement revocation

1.3.3 Inter-program dependencies and side effects

Analyzing the security of a system requires analyzing the communication patterns between programs. Limited, well-defined communication channels make this easy; unconstrained fine-grained sharing makes a mess. The Java API, unfortunately, is quite large, making it difficult to identify the points at which security must be enforced. One of the downsides of a programming language's ability to express abstractions is that it is all too easy to create complex interactions between components of a system. Consider Java's `String` class, which was originally not a final class (i.e. programs could override its methods). Because applets and browsers exchange strings at such a fine granularity, programmers could not track all of the interactions that occurred through `String` objects, leading to security problems in early versions of Java. Because of this, `String` was later made final [vdLin]. In contrast to Java's API, a traditional OS has a small and clearly defined system call interface, which forms a single point of security enforcement.

The Java programming language makes no distinction between objects shared between programs and objects private to a program. Yet, the distinction is critical for reasoning about the system's security. For example, a malicious user might try to pass a byte array holding legal bytecode to a class loader (byte arrays, like other objects, are passed by reference to method invocations), wait for the class loader to verify that the bytecode is legal, and then overwrite the legal bytecode with illegal bytecode which would subsequently be executed. The only way the class loader can protect itself from such an attack is to make its own private copy of the bytecode, which is not shared with the user and is therefore safe from malicious modification.

1.3.4 Resource Accounting

When objects are shared at a fine granularity, which programs should be charged for the objects and how can memory usage be tracked? One solution is to charge a program for the memory that it allocates. However, this breaks down when programs share data. If program A creates an object and shares it with program B, then A is charged for the object for as long as B retains a pointer to it, even if A is no longer using the object. Even after A shuts down, it may be charged for memory indefinitely.

Another solution is to charge for all the objects reachable from a program's roots. Unfortunately, this approach is dangerous when programs share abstract data types. Program A can give program B what looks like a small object, but in fact contains a private field pointing to large amounts of A's private data, and program B gets charged for all of A's data. In fact, B is still charged for the data even after program A has exited, since A's data cannot be garbage collected while B still has a reference to it. Fine-grained

sharing looks less pleasant when any shared object can act as a resource Trojan horse.

1.3.5 Dynamic loading limits optimizations

When compiled naively, safe languages tend to run much more slowly than lower level languages such as C. To reduce this performance gap, safe languages rely on relatively sophisticated optimizations. Many optimizers [DDG+96, FKR+99, Javc, MS99, WdFD+98, IKY+99, BCF+99, BK99] rely on global information (such as "this method is never overridden and may therefore be inlined") that may be invalidated as code is dynamically loaded. One way to reconcile these whole-program optimizations with dynamic loading is to undo optimizations as necessary when new code is loaded [IKY+99, HCU92, BCF+99, Javc]. Suppose a server uses the Java class `Vector`, but never overrides the methods of this class. In this situation, the server may inline method invocations on objects of type `Vector`. If an applet subsequently overrides these methods, however, the server's code must be dynamically recompiled to remove the inlining, because the applet might pass an object with the overridden methods to server, and the server might invoke one of these methods. This strategy is complicated to implement, requiring close cooperation between the compiler and run-time system. Moreover, in a language-based protection system, where multiple programs are loaded into a single environment, it penalizes one program for the actions of another program. Why should a server have to undo its internal optimizations because of code contained in a dynamically loaded applet? It is difficult to predict the performance a program when its optimization depends on other programs' code.

1.4 The task

The problems listed in the last section stem from a lack of an appropriate aggregate structure in existing language-based protection environments:

- Terminating a program requires at least some sort of aggregate structure to track which threads to stop, but this is not sufficient: terminating only threads leaves objects and code, suggesting that the aggregate structure also encompass objects and code.
- Analyzing the security of a system requires determining where the boundaries between programs are; these boundaries are determined by the structure of the system as a whole.
- Implementing revocation requires deciding which objects are sensitive enough to warrant the cost; objects shared between programs are more likely to need revocation than objects that are private to a program.
- Resource accounting requires knowing which resources belong to which program, which is again an issue of aggregate structure.
- Whole-program optimization must account for all programs loaded into a shared environment, which is too coarse: it would be better to apply such analysis at a finer granularity, so that one program's behavior does not penalize another program's performance. For this, we need some aggregate structure marking the boundaries between programs.

In this thesis, I propose an aggregate structure called a *task*, which consists of all of a program's objects, code, and threads together. Tasks are

analogous to traditional operating system processes and tasks. By dealing with objects, code, and threads together rather than in isolation, tasks make guarantees about the interaction between the three. For example, a task's threads are guaranteed only to run the task's own code—they cannot cross over a task boundary and run someone else's code. Since Java objects contain code as well as data, implementing this guarantee requires distinguishing method calls on a task's own objects (which only contain a task's own code) from method calls on other tasks' objects, which requires distinguishing one task's objects from another tasks' objects. Thus, a task's structure in terms of objects is intertwined with the task's structure in terms of threads and code.

Tasks form a framework in which revocation, termination, system analysis, resource accounting, and whole-program optimization are tractable:

- Only pointers to objects in other tasks need to be revocable; pointers to a task's own objects need not support revocation.
- When a task is terminated, its threads, code, and objects are shut down together. All pointers into the terminated task are revoked, so that the task's objects are unreachable, and can therefore be garbage collected. Although stopping the task's threads suddenly may produce damaged objects in the task, these objects are unreachable, so no live task will see them.
- Tasks establish a clear boundary between programs, so that analyzing inter-program communication is easier.
- Tasks provide a very simple resource accounting model: a task pays only for those objects that it explicitly allocates with the Java `new` operator; it is not charged for objects allocated by other tasks.

- A compiler can perform whole program optimizations at a task granularity, because dynamically loading code into one task does not affect the code of other tasks. I will call these *whole-task optimizations*.

1.4.1 Distinguishing local and remote objects

The key feature of the task model is the distinction between one task's resources and another task's resources. Actions on a task's local resources differ from actions on other task's resources. For example, the compiler or run-time system must perform revocation checks to access *remote* pointers to other tasks' objects, but not to access *local* pointers to a task's own objects. In addition, method invocations on remote pointers much switch threads so that a task's thread does not run another task's code.

The distinction between local and remote objects might be enforced by the run-time system or by the programmer. To start with, consider an implementation where the run-time system makes the distinction with no explicit directives from the programmer. Perhaps each object would be tagged with the task that allocated it, and every operation on an object would first check to see whether the object is local or remote. However, this would slow down operations on local objects and operations on remote objects.

Even if the run-time system could implement the local/remote distinction efficiently with no help from the programmer, explicit programmer involvement has a several advantages. First, remote code is more dangerous than local code (local code is trusted, while code in another task may have been written by an adversary), so the programmer should know which method calls are on remote objects, so that the method's

arguments and return values are treated with caution. Second, remote pointers are revocable, and if remote pointers aren't statically distinguished, *any pointer* is potentially revocable. This leaves Java programs on shaky ground: how can a programmer deal with a world where ordinary looking objects suddenly stop working? For instance, suppose `String` objects were revocable. Strings are often used as keys for hash tables, and a standard Java hash table is implemented as an array of linked lists, where each link holds a key/value pair. If one of the `String` keys is revoked, it will fail every time it is queried for equality. Not only does this prevent the revoked string's own associated value from being retrieved, it also prevents links following it in its list from being reached, which will probably come as a surprise to the programmer.

Rather than have the run-time system try to determine remoteness dynamically, another approach is to have the programmer implement the local/remote distinction with little or no help from the compiler and run-time system. The programmer mentally labels all code, objects and threads with a task, and then explicitly inserts extra code at method calls to remote objects to handle threads, and explicitly relinquishes pointers into terminated tasks so that they can be garbage collected. DrScheme [FFK+99] demonstrates that this strategy can work, at least in the absence of peer-to-peer communication. However, the more complex the interactions are between tasks, the more work the programmer must do to track which objects belong to which task. Furthermore, this strategy leaves the task model to the programmer's discretion—the language cannot guarantee that it is respected. Because of this, the compiler and run-time cannot rely on the task model. This rules out

whole-task optimizations, which are unsound without the guarantee that a task's threads only run a task's own code.

The strategy I adopt falls between the two approaches just described. Enforcing the task model is not left entirely to the run-time system nor to the programmer; instead, the programmer gives enough information to the run-time system for it to insert the correct revocation checks and thread switches. With this information, the run-time system guarantees that any well-typed program respects the task model. The following chapters describe implementations of the task model based on this approach. In the J-Kernel, the programmer creates explicit *capability* objects that mark the boundaries between tasks. Method invocations on these capability objects follow Java's Remote Method Invocation API and semantics, including thread switches and passing non-capability arguments by copy. Luna, on the other hand, modifies Java's type system to express the boundaries between tasks: remote pointers have special types, marked syntactically with a ~ character.

CHAPTER TWO:
THE J-KERNEL

I developed the J-Kernel as a prototype of the task model. The J-Kernel is written entirely in Java, with no native methods and no changes to the Java bytecode format or the underlying virtual machine. This allows the J-Kernel to run on commercial Java platforms, which have tended to be faster than customizable open source Java platforms such as Kaffe. It also made it easy to distribute the J-Kernel publicly, so that others could easily download it, examine it, and run it on various platforms. Finally, a Java-only implementation is easier to develop than an extension to a virtual machine, making it simpler to test the ideas of the task model.

Although the J-Kernel does not change the virtual machine, it does make some changes to the Java API, because parts of the existing API are inappropriate to the J-Kernel's task model. In particular, Java's `ClassLoader`, `Thread`, and `ThreadGroup` classes contain methods that could undermine the J-Kernel's guarantees. Therefore, these methods must be hidden from the programmer. Rather than simply eliminate these classes, the J-Kernel preserves the original Java API as much as possible by interposing its own classes [WBD+97], with the same class and method names, but with a safer implementation of the problematic methods. The J-Kernel uses its control over dynamic linking to substitute the safe classes for the dangerous classes. While this approach required no changes to the virtual machine, it was limited in some ways, which will be described later in the chapter.

The task model groups code, objects, and threads into tasks, and requires a mechanism to distinguish task-local entities from remote entities.

In Java, this requires restricting the types of objects shared between tasks in some way: if there are no restrictions, then any objects can be shared, method invocations on these objects can invoke arbitrary code in other tasks, threads can cross task boundaries, and problems with thread and task termination arise.

Nevertheless, method invocation on shared objects is a very useful communication mechanism in Java, since it naturally expresses the idea of one task exporting a service to another task. Therefore, the J-Kernel uses method invocation as the basis for its inter-task communication, but does so in a way that accommodates revocation and thread control. In order to ensure that code from a terminated task is not executed, and to ensure that threads do not directly cross task boundaries, a cross-task method invocation should perform a revocation check and a thread switch. Therefore, the J-Kernel only allows special *capability* objects to be shared between tasks, and ensures that method invocations on the capability objects perform revocation checks and simulate a thread switch.

To preserve the invariant that only capability objects are shared between tasks, method invocations on capability objects pass capability objects by reference but pass other types of objects by copy. Note that a deep copy of non-capability objects is necessary: the method invocation must recursively traverse and replicate all non-capability pointers in a data structure to copy the data. Copies, revocation checks, and thread switches increase the cost of inter-task communication when compared to ordinary Java method invocations. However, section 2.7 argues that this cost is acceptable for real applications.

2.1 Java Remote Method Invocation

The semantics for method invocations on capabilities are essentially the same as the semantics of Java's *Remote Method Invocations* (RMI) [Javd], which supports method invocations over a network. Because of this similarity, the J-Kernel bases much of its API on the Remote Method Invocation API. This section describes RMI in detail, so that the J-Kernel's interface presented in the following sections will make sense.

RMI divides objects into two categories:

- *Serializable objects* are passed by deep copy through remote method invocations. Serializable objects must belong to a class that implements the interface `Serializable`. The `Serializable` interface has no methods, but serves as a flag to let the run-time system marshal the object into a platform-independent byte stream (a "serialized" version of the object). RMI serializes an object, recursively serializing data pointed to by the object, sends the bytes over a network, and deserializes (unmarshals) the bytes on a remote computer to form a deep copy of the original object.
- *Remote objects* are passed by reference through remote method invocations. Remote objects must belong to a class that implements one or more *remote interfaces*, where a remote interface is an interface that extends the interface `Remote`. All methods declared in the remote interfaces may be invoked on a remote reference (a "stub") to a remote object. Like `Serializable`, `Remote` has no methods and serves only as a flag to the run-time system to identify remote objects, and to

identify which methods of a remote object may be invoked remotely.

If an object is neither remote nor serializable, then it cannot be passed through a remote method invocation.

Figure 2.1 shows an example of serializable objects passed through an RMI invocation:

```
class Request implements Serializable
{
    String path;
    byte[] content;
}

interface Servlet extends Remote
{
    byte[] service(Request req) throws RemoteException, IOException;
}

class FileServlet implements Servlet
{
    public byte[] service(Request req)
        throws RemoteException, IOException
    {
        // Open a file and return the data contained in the file:
        FileInputStream s = new FileInputStream(req.path);
        byte[] data = new byte[s.available];
        s.read(data);
        s.close();
        return data;
    }
}
```

Figure 2.1: Serializable object passed through RMI

Since `FileServlet` implements the remote interface `Servlet`, it can be passed from one machine to another as a reference to a remote object. Both `byte[]` and `Request` are serializable, and may be passed by copy through remote invocations on the `service` method. Notice how the semantics of remote method invocation differ from local invocation: invocations of the

`service` methods on a *local* `Servlet` object would pass and return the `Request` and `byte[]` objects by reference rather than by copy.

```
interface DispatchServlet extends Servlet
{
    void addServlet(String path, Servlet servlet) throws
RemoteException;
}

class SimpleDispatcher implements DispatchServlet
{
    Hashtable servletTable = new Hashtable();

    public void addServlet(String path, Servlet servlet)
        throws RemoteException
    {
        servletTable.put(path, servlet);
    }

    public byte[] service(Request req)
        throws RemoteException, IOException
    {
        Servlet s = (Servlet) servletTable.service(req.path);
        return s.get(req);
    }
}
```

Figure 2.2: Dispatch servlet example

A `DispatchServlet` object, shown in Figure 2.2, forwards requests to other servlets by making remote invocations on their `service` methods. A remote invocation on the `addServlet` method passes a `String` object by copy and the `Servlet` argument by reference, since `String` is serializable and `Servlet` implements remote interfaces. For example, a function with a remote reference to a `DispatchServlet` object can create and add a new servlet as follows:

```
void makeFileServlet(DispatchServlet d)
    throws RemoteException
{
    d.addServlet("foo/servlet", new FileServlet());
}
```

Figure 2.3: Remote method invocation example

This simple example hides a large amount of complexity. In particular as the new `FileServlet` object is passed from one machine to another in the remote method invocation to `addServlet`, stub objects for the `FileServlet` object are automatically generated on both machines.

2.2 J-Kernel capabilities

The combination of pass by reference and pass by copy make RMI a flexible and easy to use communication mechanism. In fact, I could have simply used Java's RMI implementation, which is a standard part of most virtual machines, as the J-Kernel's communication mechanism. However, the existing RMI implementations were orders of magnitude slower than what I desired. Furthermore, at the time I wrote the J-Kernel, the RMI specification did not support revocation. Although revocation support has since been added to RMI, in the JDK1.2's `Remote.unexportObject` method, this support is less explicit than J-Kernel's support. The key difference is that RMI implicitly creates stubs to remote objects when the remote objects are passed through a remote invocation. Because these stubs are outside the programmer's control, RMI's revocation abilities are limited: revoking access to a remote object revokes *everyone's* access to it. In other words, RMI's revocation is not selective.

By contrast, the J-Kernel makes a visible distinction between the original object and capability objects that act as remote stubs for the original

object. An object implementing remote interfaces cannot be passed directly from one task to another. Instead, the programmer first calls the static `Capability.create` method, which takes an object implementing remote interfaces and returns a *capability* (a stub object) that points to the original object, which I will call the *target object*. The capability implements all the remote interfaces implemented by the target object. Invocations on the methods of the capability's remote interface are redirected to the target object after copying the arguments and switching to the target object's task.

In addition to implementing the remote interfaces, the capability object extends the class `Capability`, which contains a public method `revoke` that revokes access to the capability:

```
public class Capability
{
    public static Capability create(Remote obj);
    public final void revoke() throws SecurityException;
    ...
}
```

Figure 2.4: The Capability class

`Capability.create` is the only mechanism available to the programmer to create instances of the class `Capability`; the J-Kernel prevents programs from directly instantiating a subclass of `Capability`. This ensures that all capability instances implement the semantics specified by `Capability.create`.

Despite the distinction between RMI stubs and J-Kernel capabilities, programs written for J-Kernel are very similar to programs designed for RMI. For example, the code shown above for the `Request`, `Servlet`, `FileServlet`, `DispatchServlet`, and `SimpleDispatcher` run on the J-

Kernel with no modification. The `makeFileServlet` function, however, must be changed to create an explicit capability, rather than relying on implicit stub creation:

```
void makeFileServlet(DispatchServlet d)
    throws RemoteException
{
    Capability c = Capability.create(new FileServlet());
    Servlet s = (Servlet) c;
    d.addServlet("foo/servlet", s);
}
```

Figure 2.5: Creating a capability

In the `makeFileServlet` example, the call to `Capability.create` expresses a task's desire to export a service to other tasks. In fact, the primary purpose of `Capability.create` is to explicitly mark the establishment of inter-task communication channels. This aids in analyzing a program's communication patterns; a simple way to look for the communication entry points into a task is to search for all the task's calls to `Capability.create`. Whereas RMI is often used to transparently spread a single protection domain across many virtual machines, the J-Kernel is designed to protect multiple domains running in a single machine. Thus, while RMI tries to hide boundaries, the J-Kernel seeks to expose boundaries.

A second purpose of `Capability.create` is to provide selective revocation. After a task creates a capability, it can revoke it by calling the capability's `revoke` method. When a capability is revoked, its pointer to its target object is overwritten with `null`, causing all future invocations of the capability's remote interface methods to fail by raising an exception. In addition, since a revoked capability no longer points to the target object, the

target object may be eligible for garbage collection, so that one task cannot use a revoked capability to prevent another task from collecting garbage.

If a task creates multiple capabilities for a single target object, it can selectively revoke access to the target object by revoking some of the capabilities while leaving other capabilities unrevoked:

```
FileServlet f = new FileServlet();
Capability c1 = Capability.create(f);
Capability c2 = Capability.create(f);
...
// Revoke c1 but not c2:
c1.revoke();
```

Figure 2.6: Selective revocation

The target object of a capability may itself be a capability [Red74]:

```
FileServlet f = new FileServlet();
Capability c1 = Capability.create(f);
Capability c2 = Capability.create(c1);
...
c1.revoke();
```

Figure 2.7: Multiple indirection

In this case, `c2` serves as a restricted version of `c1`. `c2` can be revoked without revoking `c1`, though if `c1` is revoked, `c2` is effectively revoked.

A task cannot revoke a capability created by a different task; if a task calls the revoke method on someone else's capability, the `revoke` method throws a `SecurityException`. This is a simple form of access control to the `revoke` method; the need for more complex schemes, such as access control lists or special revocation capabilities, did not arise in practice.

A third purpose of `Capability.create` is clear semantics. Capability objects pass through cross-task method invocations without

changing their identity, so that Java `instanceof` and `==` operators behave predictably when applied to capability objects. For languages like Standard ML, which have no such operators for immutable values, this is not as important of an issue.

2.3 Capability implementation

So far, this chapter has motivated the design of the J-Kernel, and the existence of `Capability.create` in particular. Now it is time to examine the implementation of `Capability.create` in detail. Consider a capability based the following remote interface:

```
interface I extends Remote
{
    int f(int x, int y) throws RemoteException;
}

class C implements I {...}

I target = new C();
Capability c = Capability.create(target);
```

Figure 2.8: Capability creation example

This call to `Capability.create` generates a stub object that implements the remote interface `I` and points to the `C` target object. This stub object is an instantiation of a class generated dynamically by the J-Kernel the first time a `C` object is passed to `Capability.create`. The dynamically generated class is depicted in Figure 2.9.

```
public class C$STUB$LRMI extends Capability implements I
{
    private C target;

    public int f(int x, int y) throws RemoteException
    {
        ThreadTaskState state = getThreadTaskState();
        Task sourceTask = state.currentTask;
        ThreadSegment callerThreadSegment =
            state.activeThreadSegment;

        try
        {
            state.switchIn(targetTask);
            int ret = target.f(x, y);
            state.switchOut(sourceTask, callerThreadSegment);

            return ret;
        }
        catch(Throwable e) {...}
    }

    ...
}
```

Figure 2.9: Capability implementation

The code shown in Figure 2.9 is written as Java source code; in reality, the J-Kernel generates Java bytecode, which is passed to a class loader directly, without requiring source-to-bytecode compilation.

The stub class `C$STUB$LRMI` belongs to the same package as `C`, so that it has access to `C` even if `C` is not public. The stub class is a subclass of `Capability`, so that it inherits the `revoke` method, and implements the `I` remote interface, which declares the method `f`. The generated implementation of `f` forwards each cross-task call to the target object in the line "`int ret = target.f(x, y);`". In addition, `f` performs three actions: it checks for revocation, it switches thread segments, and it copies arguments.

2.3.1 Revocation

The revocation check is crude but efficient: when the capability's `revoke` method is called, its `target` field is set to null. This causes the

method invocation "`target.f(x, y)`" to raise a null pointer exception, so that the capability is unusable.

2.3.2 Thread segments

Thread segment switching is a little more complicated. According to the task model, threads should not migrate across task boundaries, so a cross-task call should switch to a new thread in the callee's task, rather than running in the caller's thread. However, the expense of switching threads would increase the cost of a cross-task call by an order of magnitude on a typical virtual machine. Therefore, the J-Kernel compromises the task model to a certain extent: the caller and callee run in the same thread, but run in different *thread segments*.

Conceptually, the J-Kernel divides each Java thread into a stack of *thread segments*. In each cross-task call, the J-Kernel pushes a new segment onto this stack. When the call returns, the thread segment is popped. The J-Kernel class loader then hides the system `Thread` class that manipulates Java threads, and interposes its own `Thread` class with an identical interface but an implementation that only acts on the local thread segment. The key difference is that `Thread` modification methods such as `stop` and `suspend` act on thread segments rather than Java threads, which prevents the caller from modifying the callee's thread segment and vice-versa. This provides the illusion of thread-switching cross-task calls, without the overhead for actually switching threads. The illusion is not totally convincing, however—cross-task calls really do block, so there is no way for the caller to gracefully back out of one if the callee does not return. This problem is one drawback of a Java-only implementation, which cannot control threads and stacks at a low level

(chapter 4 describes how a customized virtual machine can implement thread switching efficiently).

```
public synchronized void switchIn(Task targetTask)
{
    // Set the thread's current thread segment and current task
    // to point to the callee:
    activeThreadSegment = null; // new segment created lazily
    currentTask = targetTask;   // thread now in the target task
}

public synchronized void switchOut(Task sourceTask,
    ThreadSegment callerThreadSegment)
{
    // Set the thread's current thread segment and current task
    // to point to the caller.
    currentTask = sourceTask;
    activeThreadSegment = callerThreadSegment;

    // Check to see if the caller's thread segment's state has
    // changed since we last saw it (for instance, the thread
    // segment may have been suspended or stopped, or the caller
    // task may have been killed entirely).  If so, take the
    // appropriate action:
    if(callerThreadSegment != null &&
       callerThreadSegment.alertFlag)
    {
        ...If segment was suspended, wait until it is resumed
        ...If segment was stopped, throw a ThreadDeath exception
    }
    if(sourceTask.dead) ...throw a TaskDeath exception...;
}
```

Figure 2.10: Thread segment implementation

Each J-Kernel thread holds a `ThreadTaskState` object that tracks which task the thread is currently running in, and which thread segment is the topmost segment on the thread's stack of segments. A capability switches thread segments by calling the method `switchIn` to move to a new thread segment, and `switchOut` to return to the original caller segment (see Figure 2.10). For efficiency, `switchIn` does not allocate a new thread segment object immediately, but instead creates one on demand if the callee later calls the method `Thread.currentThread` to get the `Thread` object that corresponds

to the new thread segment. Otherwise, no thread segment object is allocated for the callee. The `switchIn` method does, however, make a note that the thread is now running in another task, so that the J-Kernel's task termination mechanism can discover which threads are running in a task that is about to be terminated. In addition, the `switchIn` and `switchOut` methods are declared synchronized, in order to coordinate with the task termination mechanism (this will be discussed in detail below).

2.3.3 Copying arguments

In addition to checking for revocation and switching thread segments, a cross-task call makes a deep copy of all non-capability arguments and return values. The Java language already passes primitive (non-reference) types such as `int` by copy, so no special handling of these types is required. For reference types, the J-Kernel implements two copying mechanisms. First, like Java's remote method invocation, the J-Kernel can use Java's serialization to copy objects: if an argument's class is declared to implement `Serializable`, the J-Kernel serializes an argument into an array of bytes, and then deserializes the byte array to produce a fresh copy of the argument.

While serialization is convenient because many built-in Java classes implement `Serializable`, it slows cross-task calls by one to two orders of magnitude. Therefore, the J-Kernel also provides a "fast copy" mechanism, which makes direct copies of objects and their fields without using an intermediate byte array. If a class is declared to implement the interface `FastCopyGraph` or `FastCopyTree`, then the J-Kernel automatically generates specialized copy code that recursively copies the class's fields. For `FastCopyGraph` classes, the copy code keeps a hash table to ensure that no

object gets copied more than once, even if the data structure has cycles. For `FastCopyTree` classes, no hashing is performed, speeding the copying process in the case where the programmer knows that a data structure is tree shaped.

There are some subtleties in the copying process. First, the task model states that code does not migrate across task boundaries. Therefore, the copying process should not instantiate a class in the target task if the target task does not want to load the class. For copying via serialization, this restriction is enforced naturally: as an object is deserialized in the target task, Java's serialization process contacts the task's class loader to request the object's class. If the task does not want to load the class, the class loader throws an exception and the serialization process fails with an exception. On the other hand, fast copy code generated by the J-Kernel must perform an explicit check to make sure that the object's class can be loaded in the target task. This check is implemented with a hash table lookup (based on the object's `Class` object), which adds an extra overhead to the fast copying process.

Second, for objects with synchronized methods, the copying process must acquire a lock on the object as the object is copied, to make sure that the copy is a consistent snapshot of the original object. This was a bug in the release version of the J-Kernel: no lock was acquired. It also appears to be a bug (or feature) of Java serialization, which does not acquire locks during the serialization process.

Third, it matters in which task the copying takes place. In Java, programmers are allowed to write custom serialization and deserialization methods for each class, which means that code from the source task may run

during serialization, and code from the target task may run during deserialization. Therefore, the J-Kernel switches back and forth between tasks as it serializes and deserializes arguments (this is complicated by the fact that fast copy data structures may contain serializable objects). Luckily, the cost of this switching is small compared to the cost of the serialization. In addition, both serialization and fast copying may trigger dynamic class loading, which must be performed in the correct task.

In general, the J-Kernel's copying mechanisms are more complicated and ad hoc than I would have liked. The fact that copying can fail at run-time because the target task does not link to the same classes as the source task has not caused problems in practice, but seems inelegant, since a run-time mechanism is used to catch a link-time problem. These concerns motivated the development of Luna, which breaks up the copying process into smaller and simpler pieces.

2.4 Creating and terminating tasks

J-Kernel tasks are organized hierarchically, so that when a task is terminated, all of the child tasks that it spawned are terminated as well. This ensures that a malicious task cannot elude termination by creating child tasks.

When a new J-Kernel task is created, it contains no classes, objects, or thread segments. To establish state in the new task, the parent task *seeds* the new child task with an initial capability object:

```
Resolver resolver = …;
Task child = new Task(resolver);
Capability c = child.seed(seedClassName);
```

Figure 2.11: Seeding a new task

The `seed` method takes the name of the class as an argument, creates an instance of the named class in the child task, and builds a capability pointing to this new object, which is returned to the parent task. The "resolver" passed into the `Task` constructor controls the child task's linking (i.e. its mapping of class names to classes); this is discussed in detail in the next section.

The capability returned by the `seed` method serves as an initial communication channel between the parent and child, through which additional capabilities may be passed to establish more channels between the parent and child, or to establish channels between the child and arbitrary other tasks. In this respect, the child is at the mercy of the parent; if the parent does not give the child any capabilities, then the child has no way to make calls to other tasks. Thus, the parent can easily create child tasks with arbitrarily limited privileges. This encourages the principle of least privilege, and contrasts with traditional operating systems such as Unix and Windows NT, where it is relatively difficult to launch a process with arbitrarily limited privileges.

When a J-Kernel task is shut down, its thread segments are stopped and its capabilities are revoked, making the task's objects semantically unreachable. Unfortunately, the Java garbage collector cannot always tell that the task's objects are unreachable. If a task's thread segment is blocked on a cross-task call, the J-Kernel must wait until the cross-task returns to eliminate the thread segment's stack frames, since these stack frames are stuck in the middle of the stack and the J-Kernel has no direct access to the virtual machine's stack implementation. Thus, pointers in these stack frames may keep the task's objects alive in a frozen state: the objects are completely

inaccessible to any running thread, but their garbage collection is delayed. This does not affect the J-Kernel's semantics, but it does affect the machine's resource usage.

The J-Kernel uses Java's `Thread.stop` method to implement thread segment termination. This method asynchronously raises an exception in a running thread. The exception then propagates up the thread's stack, popping off stack frames until the stack is empty. Oddly, Java programs can simply catch the exception and continue execution, thwarting `Thread.stop`'s attempt to stop them. To fix this, the J-Kernel rewrites Java code so that any `catch` clauses propagate the J-Kernel's termination exception upward rather than catching it. Naturally, the J-Kernel's own run-time system classes are not subject to this restriction—they catch the thread termination exception at the boundaries between thread segments so that individual thread segments are terminated without killing a whole thread.

In order to terminate a task's thread segments, the J-Kernel must first find the task's thread segments. To speed up cross-task calls, the J-Kernel maintains no data structures mapping a task to its thread segments. In the absence of such a data structure, the J-Kernel searches for a task's thread segments by examining the top thread segment of every thread in the system. For each thread, it locks the thread's `ThreadTaskState` object so that the top thread segment is stable (this is the reason that cross-task calls require locking), and then examines the top thread segment to see if it belongs to the dying task. If so, it raises a termination exception in the thread.

The limitations of thread segments motivated my decision to develop Luna by modifying a virtual machine rather than by writing it in pure Java. Unlike the J-Kernel, Luna frees a dead task's object as soon as the next garbage

collection starts. In addition, Luna keeps a data structure holding a task's threads, so that termination does not require a search through all of the threads in the system.

2.5 Task linking

In the standard Java applet architecture, applets have very little access to Java's class loading facilities. In contrast, J-Kernel tasks are given considerable control over their own class loading. Each task has its own class namespace that maps names to classes. Classes may be local to a task, in which case they are only visible in that task's namespace, or they may be shared between multiple tasks, in which case they are visible in many namespaces. A task's namespace is controlled by a user-defined *resolver*, which is queried by the J-Kernel whenever a new class name is encountered. A task can use a resolver to load new bytecode into the system, or it can make use of existing shared classes. After a task has loaded new classes into the system, it can share these classes with other tasks if it wants, by making a `SharedClass` capability available to other tasks. Shared classes (and, transitively, the classes that shared classes refer to) are not allowed to have static fields, to prevent sharing of non-capability objects through static fields. In addition, to ensure consistency between tasks, two tasks that share a class must also share other classes referenced by that class.

Shared classes are the basis for cross-task communication: tasks must share remote interfaces and fast copy classes to establish common methods and argument types for cross-task calls. Allowing user-defined shared classes makes the cross-task communication architecture extensible; standard Java

security architectures only allow pre-defined "system classes" to be shared between tasks, and thus limit the expressiveness of cross-task communication.

Ironically, the J-Kernel needs to *prevent* the sharing of some system classes. For example, the file system and thread classes present security problems. Others contain resources that need to be defined on a per-task basis: the class `System`, for example, contains static fields holding the standard input/output streams. In general, the J-Kernel tries to minimize the number of system classes visible to tasks. Classes that would normally be loaded as system classes (such as classes containing native code) are usually loaded into a privileged task in the J-Kernel, and are accessed through cross-task communication, rather than through direct calls to system classes. For instance, our group developed a task for file system access that is called using cross-task communication. To keep compatibility with the standard Java file API, we have also written alternate versions of Java's standard file classes (in a way similar to the interposition proposed by [WBD+97]), which are just stubs that make the necessary cross-task calls. The J-Kernel moves functionality out of the system classes and into tasks for the same reasons that micro-kernels move functionality out of the operating system kernel. It makes the system as a whole extensible, i.e., it is easy for any task to provide alternate implementations of most classes that would normally be system classes (such as file, network, and thread classes). It also means that each such service can implement its own security policy. In general, it leads to a cleaner overall system structure, by enforcing a clear separation between different modules. Java libraries installed as system classes often have undocumented and unpredictable dependencies on one another. For instance, Microsoft's implementation of `java.io.File` depends on

`java.io.DataInputStream`, which depends on `com.ms.lang.SystemX`, which depends on classes in the abstract windowing toolkit. Similarly, `java.lang.Object` depends transitively on almost every standard library class in the system. In 1986, Richard Rashid warned that the UNIX kernel had "become a 'dumping ground' for every new feature or facility" [Ras86]; it seems that the Java system classes are becoming a similar dumping ground.

In general, the J-Kernel strives to accommodate communication between mutually suspicious tasks, rather than forcing one task to trust another task; cross-task calls through capabilities, for example, work even in the absence of trust. However, the J-Kernel's shared class mechanism is asymmetric: one task creates a class, and then other tasks import the class, so that these other tasks must trust the first task's definition of the class. This means that two mutually suspicious tasks cannot introduce new shared classes into the system by themselves, because neither task trusts the other task to define the mutually shared classes correctly. Instead, they must rely on a third party that they both trust to define the classes. While this is not ideal, there were no obvious alternatives. If two mutually suspicious tasks both have their own definitions of a class that they want to share, it is undecidable in general whether these definitions are equivalent (since classes contain code). The system could decide the equivalence by requiring that the definitions be byte-for-byte equal, but this leads to versioning problems.

2.5.1 Creating resolvers

All linking in the J-Kernel works through user-defined *resolvers*. To create a resolver, a task implements the `resolveClassName` method of the J-Kernel interface `Resolver`, shown below. This method takes the name of a

class as an argument and returns either a byte array containing new bytecode, or a `SharedClass` object from another task (the return type of `Object` is a crude way to represent a union of these two types; in retrospect, the J-Kernel should have used an additional class to represent this union). When a new task is created, it is given an initial resolver. Afterwards, a running task can add more resolvers in order to dynamically load more code.

```
public interface Resolver extends Remote {
    Object resolveClassName(String name) throws RemoteException;
}
```

Figure 2.12: Resolver interface

The resolver interface is considerably simpler and clearer than Java's `ClassLoader` API, and it shields the programmer from the dangers of raw class loaders. Unlike class loaders, resolvers do not have to load and link classes in separate steps, nor do they have to keep a table of previously loaded classes in order to handle multiple requests for the same class (the J-Kernel will only query a task's resolver once for each class it needs). In addition, the J-Kernel comes with several built-in resolver implementations for loading classes from the file system or the network (for tasks that have access to these resources), building tables of name-to-class mappings, and combining resolvers together.

Although resolvers express linking at a higher level than class loaders, in practice we found that this level was not high enough. People using the J-Kernel had more difficulties with linking than with any other J-Kernel mechanism. The most common error was to instantiate a class separately in two tasks, rather than sharing it. One common version of this problem was remembering to share some class A, but then forget to share a class B

referenced by A. The first major difficulty in the J-Kernel was simply diagnosing this kind of problem, since errors are not detected until run-time, often after several tasks are created. Sometimes the J-Kernel's linker detected the error and raised an exception. At other times, the error was detected by a failed cast. Detailed error messages were essential; the J-Kernel reports errors like "`SharedClassException: Apple refers to Bear, Bear refers to Cross, Cross has static fields`" and "`SharedClassException: if two tasks share the class Launch, then they must also share the class LaunchData`." The resulting exceptions are often thrown across task boundaries, and it was difficult to copy them from task to task without losing the original stack trace. Sometimes Java's run-time would throw away the message contained in the exception as it propagated through Java's class loading functions.

To minimize linking problems, I wrote a small module language to express class sharing statically. In this language, tasks declare the classes that they create and the shared classes they export to other tasks or import from other tasks. A simple command-line tool reads these declarations, examines the class files named in the declarations to check that they obey the J-Kernel's sharing rules, and collects the class files into an archive file, similar to a Java JAR file, but which also contains linking information. A special resolver then reads these archive files when the J-Kernel runs. The advantage of this approach is that the command-line tool catches many potential errors statically. Ideally, it would catch all linking errors, but Java's reflection mechanisms and casts make this difficult.

2.6 J-Kernel micro-benchmarks

To evaluate the performance of the J-Kernel mechanisms we measured a number of micro-benchmarks on the J-Kernel as well as on a number of reference systems. Unless otherwise indicated, all micro-benchmarks were run on 200Mhz Pentium-Pro systems running Windows NT 4.0 and the Java virtual machines used were Microsoft's VM (MS-VM) and Sun's VM with Symantec's JIT compiler (Sun-VM). All numbers are averaged over a large number of iterations.

2.6.1 Null LRMI

Table 2.1 dissects the cost of a null cross-task call (null LRMI) and compares it to the cost of a regular method invocation, which takes a few tens of nanoseconds. The J-Kernel null LRMI takes 60x to 180x longer than a regular method invocation. With MS-VM, a significant fraction of the cost lies in the interface method invocation necessary to enter the stub. Additional overheads include the synchronization cost when changing thread segments (two lock acquire/release pairs per call) and the overhead of looking up the current thread. Overall, these three operations account for about 70% of the cross-task call on MS-VM and about 80% on Sun-VM. Given that the implementations of the three operations are independent, we expect significantly better performance in a system that includes the best of both VMs.

Table 2.1: Cost of null method invocations

Operation	MS-VM	Sun-VM
Regular method invocation	0.04µs	0.03µs
Interface method invocation	0.54µs	0.05µs
Thread info lookup	0.55µs	0.29µs
Acquire and release lock	0.20µs	1.91µs
J-Kernel LRMI	2.22µs	5.41µs

To compare the J-Kernel LRMI with traditional OS cross-task calls, Table 2.2 shows the cost of several forms of local RPC available on NT. *NT-RPC* is the standard, user-level RPC facility. *COM out-of-proc* is the cost of a null interface invocation to a COM component located in a separate process on the same machine. The communication between two fully protected components is at least a factor of 3000 from a regular C++ invocation (shown as *COM in-proc*).

Table 2.2: Local RPC costs using NT mechanisms

Form of RPC	Time
NT-RPC	109µs
COM out-of-proc	99µs
COM in-proc	0.03µs

2.6.2 Threads

Table 2.3 shows the cost of switching back and forth between two Java threads in MS-VM and Sun-VM. The base cost of two context switches between NT kernel threads (*NT-base*) is 8.6µs, and Java introduces an additional 1-2µs of overhead. This confirms that switching Java threads during cross-task calls would add a significant cost to J-Kernel LRMI.

Table 2.3: Cost of a double thread switch using regular Java threads

NT-base	MS-VM	Sun-VM
8.6µs	9.8µs	10.2µs

2.6.3 Argument Copying

Table 2.4 compares the cost of copying arguments during a J-Kernel LRMI using Java serialization and using the J-Kernel's fast-copy mechanism. By making direct copies of the objects and their fields without using an intermediate Java byte-array, the fast-copy mechanism improves the performance of LRMI substantially— more than an order of magnitude for large arguments. The performance difference between the second and third rows (both copy the same number of bytes) is due to the cost of object allocation and invocations of the copying routine for every object.

Table 2.4: Cost of argument copying

Number of objects and size	MS-VM LRMI w/serialization	MS-VM LRMI w/fast-copy	Sun-VM LRMI w/serialization	Sun-VM LRMI w/fast-copy
1 x 10 bytes	104µs	4.8µs	331µs	13.7µs
1 x 100 bytes	158µs	7.7µs	509µs	18.5µs
10 x 10 bytes	193µs	23.3µs	521µs	79.3µs
1 x 1000 bytes	633µs	19.2µs	2105µs	66.7µs

In summary, the micro-benchmark results are encouraging in that the cost of a cross-task call is 50x lower in the J-Kernel than in NT. However, the J-Kernel cross-task call still incurs a stiff penalty over a plain method invocation. Part of this is due to the J-Kernel's pure Java implementation; the following chapters show that a customized virtual machine performs faster cross-task calls.

2.7 Application benchmarks

A couple of significant applications have been written for the J-Kernel. Chi-Chao Chang developed a plug-in for Microsoft's IIS web server to extend the server with J-Kernel servlets. Dan Spoonhower developed an integrated web and telephony server to support web/telephony servlets, such as a voice-mail servlet with an HTTP interface. More information about these systems is contained in Hawblitzel et al [HCC+98] and Spoonhower et al [SCH+98]. This section summarizes the key performance results from these papers.

To quantify the impact of the J-Kernel overheads in the performance of the HTTP server, several simple experiments measure the number of documents per second that can be served by Microsoft's IIS and J-Kernel running inside IIS. The hardware platform consists of a quad-processor 200MHz Pentium-Pro (results obtained on one- and two-processor machines are similar). The parameter of the experiments is the size of document being served. All three tests follow the same scenario: eight multithreaded clients repeatedly request the same document. IIS serves documents in a traditional way—by fetching them from NT's file cache, while the J-Kernel uses a servlet to return in-memory documents. Table 2.5 shows that the overhead of passing requests into and out of the J-Kernel decreases IIS's performance by 20%. Additional measurements show that the ISAPI bridge connecting IIS to the J-Kernel accounts for about half of that performance gap and only the remainder is directly attributable to the J-Kernel execution time.

Table 2.5: Server throughput

Page size	IIS throughput (pages per second)	IIS+J-Kernel throughput (pages per second)
10 bytes	801	662
100 bytes	790	640
1000 bytes	759	616

The web/telephony server experiments were carried out on a 300MHz Pentium II, running Windows NT 4.0. The Java virtual machine used was Microsoft's VM version 1.1, with SDK 2.01. Table 2.6 shows two benchmarks. The first tests the performance of transferring voice-quality audio through the system. Two servlets hold a "conversation" by exchanging two-second chunks of audio data over a phone line. Each servlet records the data sent by the other onto disk. The "initiator" and "responder" columns of the table show the performance of the two servlets. The second test, shown in the "voice servlet" column, measures the cost of making an HTTP request to a voice-mail servlet to check for mail. This includes the cost of contacting a separate authentication servlet to authenticate the request, and the cost of creating an HTML response to the request.

Table 2.6: Web/Telephony server performance

	Initiator	Responder	VoiceServlet
Elapsed time [ms]	40000	40000	67
CPU time [ms]	1503	1062	18.5
Cross-domain calls	3085	2671	8
Cross-domain data transfer [B]	962304	984068	3596
Cross-domain calls overhead [ms]	9.37	8.71	0.037
Cross-domain copy time [ms]	2.91	2.58	0.01
Time/call [ms]	0.0030	0.0033	0.0046
Bytes/call	312	368	450
Copy overhead/call	31%	30%	27%
Copy overhead [bytes/us]	331	381	360
Cross-domain call overhead	0.62%	0.82%	0.20%

In all cases the overheads of crossing tasks (which include the cost of copying arguments and then return values) are below 1% of the total consumed CPU time. On the average, crossing tasks costs between 3-4.6•s and an average call transfers between 312 (*Initiator*) to 450 (*VoiceServlet*) bytes. The cost of copying data accounts for 27-31% of an average cross-task call. This suggests that overheads of crossing tasks in real applications when no data transfer is involved are between 2.1-3.3•s. The value can be contrasted with the cost of a null cross-task call, measured in an isolated tight loop on a 300MHz machine: 1.35•s.

2.8 Conclusions

The J-Kernel demonstrates the feasibility of the task model, and shows that the overheads introduced by enforcing the task model are small relative to the performance of Java applications. Just as importantly, it demonstrates that tasks can be built with a reasonable programming model—although Java's RMI interface is hardly elegant, it does allow the programmer to express and enforce abstractions at a high level.

The J-Kernel's Java-only implementation was convenient for a prototype, but probably insufficient for a real-world product. It would be difficult to do very accurate accounting of task resource consumption (see Czajkowski et al [CvE98] and Back et al [BHL00] for comments on this), and the J-Kernel had to compromise its thread switching semantics because it did not have low-level access to threads and stacks. In addition, built-in classes such as `Class`, `Object`, and `String` have behaviors and dependencies that are hard to work around. While bytecode generation and modification can overcome some problems, such as preventing code from catching thread

termination exceptions, it slows class loading and is often awkward to implement. Experience with the J-Kernel motivated the implementation of Luna, described in the next chapter.

CHAPTER THREE:
LUNA DESIGN

Luna is similar in spirit to the J-Kernel, but makes several improvements. First, Luna strips away much of the complexity associated with the J-Kernel's RMI-based interface, replacing it with a relatively simple and orthogonal type-based interface. The J-Kernel's capabilities are black boxes—the programmer annotates classes as `Remote`, `Serializable`, `FastCopyTree`, and `FastCopyGraph`, calls `Capability.create`, and after some magic bytecode generation, the J-Kernel returns a capability suitable for inter-task communication. Luna makes inter-task communication more transparent and flexible by breaking the J-Kernel's black box mechanisms into simpler primitives. The validity of these primitives is enforced statically by Luna's type systems rather than by the J-Kernel's dynamic mechanisms.

Second, the J-Kernel only lets tasks share capabilities; other types of objects must be passed by copy, making it impossible to share arrays and object fields directly. Luna introduces a limited form of sharing for these types, so that programs can share data directly. Nevertheless, Luna still relies heavily on copies; it is often easier to copy a remote object than to manipulate remote fields and arrays directly.

Luna's third improvement over the J-Kernel is in its implementation. In writing the J-Kernel's run-time system, I was frustrated by Java's inability to manipulate threads, stacks, locks, and heaps at a low level, and in general I was disappointed by the performance of commercial just-in-time compilers. Therefore, I implemented Luna as an extension to Marmot [FKR+99], a highly

optimizing virtual machine from Microsoft Research. Marmot was designed as a "way-ahead-of-time" compiler rather than a just-in-time compiler; it performs whole-program optimizations but does not support dynamic loading. This fits in with Luna's preference of static analysis over run-time analysis—although Luna adds dynamic loading and unloading to Marmot, it does so at the granularity of tasks rather than classes, and uses Marmot's whole-program optimizations to statically optimize tasks when they are loaded.

This chapter describes Luna's design in detail. It introduces Luna's remote pointer types, which allow tasks to both share data structures and implement J-Kernel style capabilities, and describes how the semantics of some operations, such as copying, differ from the J-Kernel's semantics. Luna's implementation and performance are discussed in the next chapter.

3.1 Remote pointers and revocation

Luna extends Java's type system with *remote pointers*, which are analogous to J-Kernel capabilities—remote pointers can point to objects in other tasks, while ordinary, local pointers can only point to a task's own objects. However, while J-Kernel capabilities only support a limited set of remote interface types, there is a remote pointer type corresponding to every local pointer type. A remote pointer type is syntactically represented as a local pointer type followed by a ~ character. Figure 3.1 shows the syntax for Luna's type system; standard Java types are written in plain text and the new types are written in bold.

```
Type = PrimitiveType | ReferenceType | ReferenceType~
PrimitiveType = boolean | byte | short | int | long
              | char | float | double
ReferenceType = ClassType | InterfaceType | Type[]
```

Figure 3.1: Luna type system

In order to preserve the advantages of safe language protection (fine-grained sharing, low cross-task call overheads, simple capability-based access control, and enforcement of abstract data types), remote pointers support the same operations as local pointers: field/array element access, method invocation, synchronization, equality testing, casting, and `instanceof` testing, although most of these operations have different semantics and performance for remote pointers.

The key difference between remote pointers and local pointers is revocation. Luna's task model requires that remote pointers into a task be revoked when the task is terminated, but revocation is also useful at a finer granularity. To realize these uses, Luna gives the programmer a special handle with which to control access to remote pointers. This handle is a Java object called a *permit*. A permit is allocated in an unrevoked state, and can later be revoked:

```
public class Permit {
    public Permit(); // constructor to allocate an unrevoked permit
    public void revoke(); // method to revoke the permit
    ...
}
```

Figure 3.2: The Permit class

A remote pointer is implemented as a two-word value (like Java's `long` and `double` types) that consists of a local pointer paired with a permit. The `@` operator converts a local pointer into a remote pointer:

```
Permit p = new Permit();
String s = "hello";
String~ sr = s @ p;
```

Figure 3.3: The @ operator

Once a task has used the `@` operator to create a remote pointer, it can pass the remote pointer to other tasks, which can use them until the remote pointer's permit is revoked. Each operation on a remote pointer automatically performs a run-time access check of the remote pointer's permit: the expression "`sr.length()`" will evaluate `sr`'s length if `p` is unrevoked, and raise an exception if `p` is revoked. Permits can selectively revoke access to data: if permit `p1` is revoked while `p2` is not, then `s` is accessible through the remote pointer `(s @ p2)` but inaccessible through the remote pointer `(s @ p1)`. Note that there is no way to decompose a remote pointer into its two parts: the local pointers to `s` and `p` cannot be extracted from `sr`; this prevents other tasks from gaining direct access to them. In other words, a task's access to another task's data is always mediated by a permit. Saltzer and Schroeder argue that this sort of "complete mediation," which makes it impossible for a task to circumvent the system's access control mechanisms, is one cornerstone of a secure system [SS75][WBD+97].

When a task is terminated, all the permits created by the task are revoked, which is similar to how the J-Kernel revokes all a task's capabilities when the task is terminated. As in the J-Kernel, this mass revocation makes all of a task's objects unreachable and therefore garbage collectable.

In the J-Kernel, each capability is individually revocable. This is often too fine a granularity. To enforce the principle of least privilege, it is prudent to revoke access to capabilities after completing an interaction with another task. However, if the interaction involves a large number of capabilities, this

forces the programmer to remember to revoke each capability individually. In Luna, permits handle this burden: a programmer simply uses a single permit to create all the remote pointers involved in the interaction, and then revokes all the remote pointers by revoking one permit. This *aggregation* of access control is an important feature of sharing data structures, which is discussed in more detail in the next section.

3.2 Sharing data with remote pointers

Many operations on remote pointers transfer values across task boundaries. The type system must ensure that these transfers do not create local pointers that cross task boundaries, since only remote pointers are allowed to cross task boundaries. Figure 3.4 shows the typechecking rules for local and remote field operations. Because Java is an imperative language, there are two judgments to typecheck an expression: the judgment "`e:T`" says that an expression `e` has type `T`, and the judgment "`e` is an l-value" says that the expression `e` can be assigned to. The statement "`f` is accessible" means that an expression is allowed access to field `f` according to Java's rules for `private`, default, `protected`, and `public` access controls. The rules for reading and writing remote array elements, passing arguments to remote methods, and receiving return values from remote methods are similar to the rules for remote field operations.

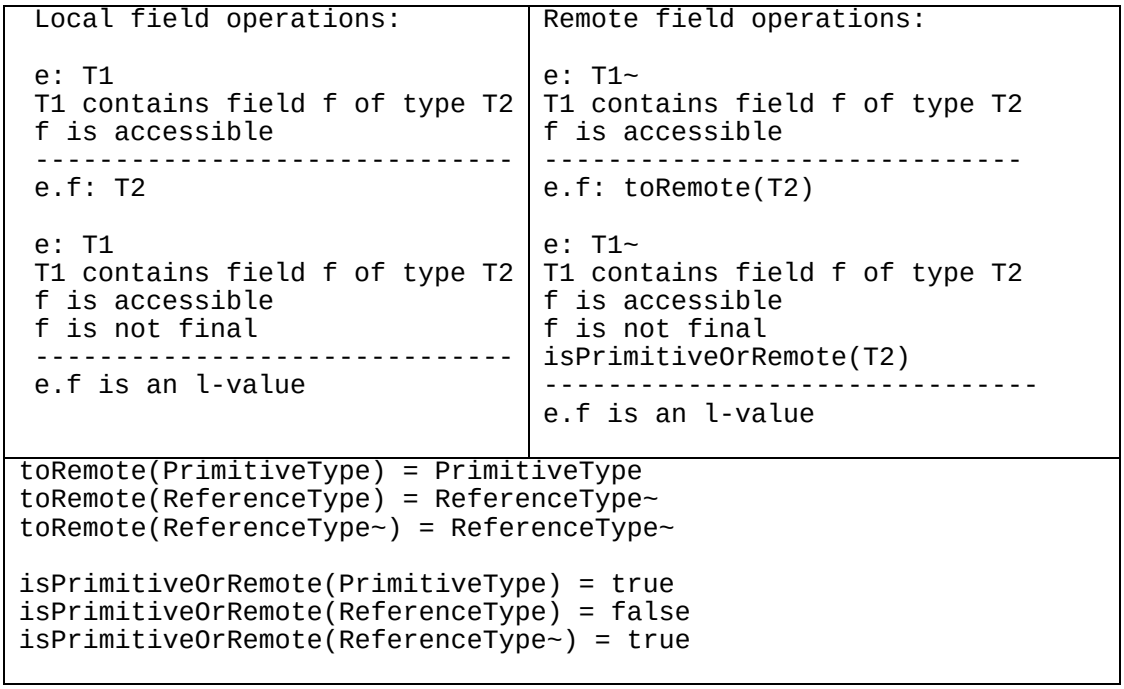

Figure 3.4: Typechecking rules for field operations

These typechecking rules allow primitive types and remote pointers to move freely across task boundaries through field operations on remote pointers. However, if task A has a remote pointer `e` to an object in task B, and `e` contains a field `f` holding a local pointer of type `T`, then A cannot view this pointer with local type `T`, since the pointer is local to B rather than A. From A's perspective, the pointer is remote, so the expression "`e.f`" has type `T~`, not type `T`. Furthermore, A cannot write to `e.f` at all: A cannot write one of A's local pointers to `e.f`, because `e.f` is supposed to contain pointers local to B, not A, and A cannot write a remote pointer to `e.f`, because this remote pointer might point to an object in any task. Although the assignment expression "`e.f = e.f`" would not violate the task model at run-time, Luna's type system is not strong enough to prove this.

Remote pointers cannot be used interchangeably with local pointers. For instance, a hash table object expecting keys of type `Object` cannot be

passed a remote pointer (which has type `Object~`). This forces the programmer to either design a new type of hash table that can accommodate remote keys (and can deal with their revocation robustly), or, more commonly, to make a local copy of a remote object's data and use that as the hash table key. Similarly, remote pointers, like Java's primitive types, must be boxed by the programmer to be placed in Java's standard container classes, such as `Vector`, that store `Object`s (this also serves a pragmatic implementation purpose, because remote pointers are a different size than local pointers, and are treated differently by the garbage collector).

The function below shows an example of using remote data structures. Notice that according to the rules above, the expression "`list.next`" has type `List~`, not type `List`. At run-time, "`list.next`" evaluates to a remote pointer containing the *same permit* as the remote pointer `list` contains. Thus, a single permit controls access to the entire list, not just the first element. If this permit is revoked, access to the entire list is immediately revoked, and the revocation is enforced by the dynamic access control checks that Luna inserts in the expressions "`list.i`" and "`list.next`". This *aggregate access control* allows tasks to share data structures and still have a single point of access control over the sharing—even if the function below scans through the list multiple times, or scans through the list in a different order, it cannot circumvent this access control.

```
class List {
    int i;
    List next;
}
void replace(List~ list, int from, int to) {
    while(list != null) {
        if(list.i == from) list.i = to;
        list = list.next;
    }
}
```

Figure 3.5: Using remote data structures

The `replace` function shown above works on remote pointers, not local pointers. Does the programmer have to write two copies of every function, one for remote pointers and one for local pointers? In general, no—Luna was designed to be used like the J-Kernel (which in turn was designed to be used like RMI), and most interaction between tasks occurs through method invocations on remote objects rather than through directly manipulation of remote fields. Although one of Luna's goals is to make this sort of direct manipulation possible when necessary, it is the exception, rather than the rule. Unless the performance of the `replace` function above were particularly critical, a programmer would typically use a `replace` function that acts on local pointers, and either make a cross-task call into the task that owns the list or make a local copy of the list and run `replace` on that.

3.3 Method invocations on remote pointers

Although the direct sharing mechanisms in the previous section are very different from J-Kernel's RMI-based communication, most interactions between Luna tasks are still based on J-Kernel-style remote method invocation with copies. However, Luna gives the programmer considerable flexibility in implementing these method invocations, and allows the programmer to add direct sharing where RMI is insufficient. Like J-Kernel capability invocations,

method invocations on Luna remote pointers check for revocation and transfer control from one task to another. In fact, Luna genuinely switches threads in a cross-task method invocation, which is more powerful than the J-Kernel's thread segment mechanism. Luna does not automatically copy objects from one task to another, though. Instead, the programmer specifies how and when data is transferred across task boundaries.

Suppose we want to write a server in Luna that runs the `FileServlet` class from the previous chapter (see Figure 3.6), without changing the basic Servlet interface that it is built over in any essential way (a common situation, since many standard Java API's are in wide use and cannot be changed lightly):

```
class Request implements Serializable {
    String path;
    byte[] content;
}

interface Servlet {
    byte[] service(Request req);
}

class FileServlet extends Servlet
{
    public byte[] service(Request req) {
        // Open a file and return the data contained in the file:
        FileInputStream s = new FileInputStream(req.path);
        byte[] data = new byte[s.available];
        s.read(data);
        s.close();
        return data;
    }
}
```

Figure 3.6: FileServlet example

Suppose the server receives a request, creates a `Request` object, and then wants to dispatch this request to a `FileServlet` running in another task. The server cannot pass a local pointer to the `Request` object to the

`FileServlet`'s task, because local pointers cannot be passed across task boundaries. Instead, the server creates a remote pointer and passes this to the servlet task. However, the `FileServlet` class, as written above, expects a local pointer, not a remote pointer. To satisfy this expectation, an intermediate step is needed to construct a local copy of the request. To implement this, it is helpful to first build a function that copies a remote `Request` object.

3.3.1 Copying remote objects

In the J-Kernel, copies are generated automatically by the run-time system, which cannot be customized by the user. By contrast, in Luna, the programmer writes copy routines, using operations on remote fields and remote arrays as building blocks. A copy routine recursively traverses a remote data structure, allocating new local objects to match the old remote objects. This recursion bottoms out with primitive types, which can be passed directly across task boundaries. Assuming that a method `String.copy` was already written using these techniques, the following method produces a local `Request` object given a remote `Request` object. For convenience, this method is placed in the class `Request`, although it could be placed anywhere:

```
class Request {
    ...
    static Request copy(Request~ from) {
        Request to = new Request();
        to.path = String.copy(from.url);
        to.content = new byte[from.content.length];
        for(int i = 0; i < from.content.length; i++)
            to.content[i] = from.content[i];
    }
}
```

Figure 3.7: Copying a Request object

Since copy functions are written entirely in Luna code, programmers can customize data structure copy mechanisms. A program is often able to employ knowledge about characteristics of a data structure, such as alias information, to improve the copy function. Furthermore, a program may copy a data structure lazily rather than immediately, copy data into preallocated buffer space, or never copy the data at all. On the other hand, Luna's more flexible approach to sharing sacrifices one feature of the J-Kernel: after a J-Kernel cross-task call returns, the caller knows that the arguments to the call were copied, and does not have to worry about data leaking through side effects on shared data structures. By introducing direct sharing of data, Luna introduces the dangers of such side effects.

In the J-Kernel, copies fail if the target task does not support the class of the object being copied. Luna takes a different approach. Suppose `String` has a subclass `ColorString`:

```
class ColorString extends String {
    int color;

    static ColorString copy(ColorString~ list) {...}

    ...
}
```

Figure 3.8: Subclass example

Now suppose that task A passes an object of type `ColorString` to task B, which links to the class `String` but does not link to the class `ColorString`. Since B does not know about `ColorString`, it will call `String.copy` to copy the string rather than `ColorString.copy`. Calling `String.copy` on an object of type `ColorString` produces a `String` rather than a `ColorString`—only the superclass data is copied, and the extra data in the subclass gets "sliced" away.

This may set off a few warning bells for experts in object orientation; slicing occurs sometimes in C++ and is considered a bad idea, because the sliced copy throws away the subclass's overridden methods and thus behaves differently from the original object. However, this is appropriate for inter-task communication, because B wants to copy the *data* from A's string, not the *behavior* of A's string—the behavior defined by A's code is potentially dangerous, and the task model guarantees that B only runs B's own code, and does not accidentally import A's code at run-time. Consider the security holes in early versions of Java caused by strings with overridden methods, which forced Java's designers to make the class `String` final [vdLin]. One could argue that strings must be final anyway for performance reasons, since finality makes it easier to inline method invocations in ordinary Java implementations. In Luna, though, there is no reason to make `String` final, other than backwards compatibility with the current Java API: the security problem is solved by copying and slicing strings passed across task boundaries, and the performance problem is solved with whole-task optimization.

Java's problem with strings demonstrates the tension between security and object orientation, and Luna's approach to copying favors security over object orientation. However, Luna's inter-task communication mechanisms do not throw away object orientation entirely—method invocations on remote objects are dispatched in Java's normal object-oriented fashion.

3.3.2 Combining copies and method invocation

Using the copying function from the previous section, the following class serves as an intermediary between the server and the standard `Servlet` interface:

```
class RemoteServlet
{
    Servlet servlet;

    remote byte[] service(Request~ req1)
    {
        // Construct a local copy of the request
        Request req2 = Request.copy(req1);

        // Pass the local copy of the request to the servlet
        return servlet.service(req2);
    }
}
```

Figure 3.9: Intermediary to Servlet interface

A `RemoteServer` object runs in the servlet's task and transfers requests from the server to the servlet. The server makes a method invocation on a remote pointer to this object, passing a remote pointer to the `Request` object:

```
class Server {
    ...
    byte[]~ dispatch(RemoteServlet~ servlet, Request req) {
        Permit p = new Permit();
        byte[]~ response = servlet.service(req @ p);
        p.revoke();
        return response;
    }
}
```

Figure 3.10: Calling a RemoteServlet

Rather than allowing all methods to be called remotely, Luna only allows remote method calls on methods that are declared `remote`. Like Java's

`public` and `private`, `remote` is simply an access control flag—it does not change the semantics or implementation of the method. The `remote` flag helps a programmer define a small set of entry points into a task, because not every method should support remote method calls. For example, Java defines a `clone` method in `Object`, the superclass of all classes. Without extra access control, task A could attack task B by repeatedly calling `clone` on one of B's objects until B runs out of memory. I could have also added a flag for access to remote fields for the sake of orthogonality, but this did not seem necessary. Moreover, to really be orthogonal, arrays would also need such a flag, but this would force Luna to have two different and incompatible array types.

3.4 Type system design

Luna's extension to Java's type system is fairly modest—there is just one new type, the remote pointer. However, during the development of Luna I experimented with more elaborate designs. In fact, Luna evolved from earlier, more unwieldy designs towards its current, relatively simple state. This section describes some of the design space that surrounds Luna's type system. This plays a role in Luna's implementation, since Luna's intermediate language uses a more sophisticated type system than Luna's source language uses.

Luna's type system was originally inspired by the *regions* of Tofte et al [TT94], a memory management mechanism based on stacks of regions, where a region is a list of blocks of memory, and memory deallocation is performed over entire regions at once. Statically, the program directs the run-time system to allocate each object in a particular region, and the type of each object is annotated with the region in which it is allocated, so that a type system can

ensure that an object is not accessed after its region is deallocated. Since there may be an unbounded number of regions at run-time, a program uses *region variables* to refer to regions statically. Translating these ideas from regions and region variables to permits and permit variables, the `replace` function shown earlier is expressed, loosely mixing formal syntax and Java syntax, as:

```
Λρ.void replace(permit{ρ} p, List{ρ} list, int from, int to) {
    while(list != null) {
        if(list.i == from) list.i = to;
        list = list.next;
    }
}
```

Figure 3.11: Permit variables

In this representation, the `replace` function is quantified over a permit variable ρ, and the two-word remote pointer to the list is split into its two components, a permit of type `permit{ρ}` and a pointer of type `List{ρ}`. The permit variable connects the two components together. This representation has two advantages over Luna's source-level representation. In the original Luna source code, the expressions `list` and `list.next` both have type `List~`, which expresses the fact that both `list` and `list.next` are remote pointers, but does not express the fact that at run-time, `list` and `list.next` are protected by the same permit. By contrast, with permit variables, the expressions `list` and `list.next` both have type `List{ρ}`, which explicitly signals that they are both protected by the same permit (the permit associated with ρ). The second advantage of this representation is that `list` is a one-word rather than two-word value, so that it is clear that an assignment like "`list = list.next`" need only transfer one word, not two. Because of these advantages, Luna's low-level typed intermediate language

explicitly splits local variables holding remote pointers into permit and local pointer components connected by permit variables.

In fact, these ideas can be generalized to encompass all pointers, both local and remote, throughout the entire type system. Hawblitzel et al [HvE99] describes a formalization of this idea, based on formalizations of regions [TT94, CWM99]. This system makes no distinction between local and remote pointers—all pointers are annotated with a permit, and there is no analogue to Luna's @ operator. When an object is allocated, it is immediately tagged with a permit, and it retains this permit annotation for its entire lifetime. This is simpler, but less flexible, than Luna's approach, where an object is first allocated with a local pointer type and then coerced with the @ operator to a remote pointer type. In particular, the only way to revoke access to an object is to deallocate it entirely; there is no way to selectively revoke access to an object.

[HvE99] proposes a couple of mechanisms for extending the type system with selective revocation, both of which are more flexible than Luna's mechanism. Consider how the J-Kernel can chain multiple capabilities together, in order to restrict the access rights of an existing capability:

```
FileServlet f = new FileServlet();
Capability c1 = Capability.create(f);
Capability c2 = Capability.create(c1);
```

Figure 3.12: Chaining capabilities

Luna's permits do not support this idiom. The @ operator applied once to a local pointer converts a local pointer type A to a remote pointer type A~, but there is no operator to further restrict the remote pointer type. It would not be too difficult to modify Luna's type system so that applying multiple

permits to an object adds multiple tildes, so that "`(new A() @ p1) @ p2`" has type `A~~`, but this is very awkward—now the programmer has to write different methods to deal with arguments of type `A~`, `A~~`, `A~~~`, etc. It would be much better if both "`new A() @ p1`" and "`(new A() @ p1) @ p2`" had type `A~`, so that they could be used interchangeably.

The remote pointer "`(new A() @ p1) @ p2`" is a pointer that can only be used if both `p1` and `p2` are still unrevoked. Suppose we define a new permit `p3` whose lifetime is the intersection of `p1`'s lifetime and `p2`'s lifetime, so that it is revoked as soon as either `p1` or `p2` is revoked. Then "`(new A() @ p1) @ p2`" is equivalent to "`new A() @ p3`". Based on this idea, a run-time system could implement the expression "`(new A() @ p1) @ p2`" by automatically allocating a permit `p3` whose lifetime is a sublifetime of `p1` and `p2`.

[HvE99] proposed a variation on this: if we somehow knew that `p2`'s lifetime was contained in `p1`'s lifetime, then we can simply implement "`(new A() @ p1) @ p2`" as "`new A() @ p2`" without allocating a new permit. If permits are arranged hierarchically, so that a child permit is revoked when its parent is revoked, then the type system can track the hierarchical relationship between permits and coerce "`new A() @ p1`" to "`new A() @ p2`" if `p2` is a descendent of `p1`. Tracking this information complicates the type system, which discouraged me from adding this to Luna.

CHAPTER FOUR:
LUNA IMPLEMENTATION AND PERFORMANCE

Luna is implemented as an extension to Marmot [FKR+99], an optimizing virtual machine, on the x86 running under Windows NT. Whereas the J-Kernel's performance was limited by its portable, pure Java implementation, Luna's implementation favors speed and power over simplicity and portability. It is designed to test both the maximum possible performance and the most complete possible semantics. Key features of the implementation include:

- optimized thread management for cross-task method invocation,
- special "caching" optimizations for repeated accesses to remote data,
- the use of Marmot's whole-program optimizations in an environment that supports dynamic loading, and
- garbage collection that treats remote pointers specially.

The main drawback of this implementation is its difficulty. Marmot is a large piece of software, and Luna's changes to Marmot are pervasive, intricate, and subtle. The second major drawback is portability—Luna runs on only one virtual machine, which is not distributable due to Marmot's license restrictions.

4.1 Extended bytecode format

Java implementations are split into two components, a compiler that translates Java source code into Java bytecode, and a virtual machine that executes Java bytecode. Luna is implemented in the same way. The Luna

source compiler, a simple extension to the publicly available guavac compiler, translates Luna code into Luna bytecode, which extends the Java bytecode format with instructions to handle remote pointers. Luna's bytecode format is backwards-compatible with Java's format, so that ordinary Java code will run on Luna's virtual machine. Luna code will not run on an ordinary Java virtual machine, though, because Luna's extensions require changes to the virtual machine's run-time system.

In general, there is one new Luna remote pointer instruction for each standard Java local pointer instruction, as shown in Table 4.1(Java's bytecode format uses the "a" character to indicate a local pointer, or, somewhat confusingly, an array; Luna uses the "r" character to indicate a remote pointer):

Table 4.1: Luna remote pointer instructions

Local pointer operation	Remote pointer operation	Description
`getfield, putfield`	`getfield_r, putfield_r`	read/write object fields
`invokevirtual, invokeinterface`	`invokevirtual_r, invokeinterface_r`	method invocation
`iaload, iastore, laload, ...`	`iaload_r, iastore_r, laload_r, ...`	read/write array elements
`arraylength`	`arraylength_r`	get array length
`acmp`	`arcmp, racmp, rrcmp`	compare pointers
`checkcast, instanceof`	`checkcast_r, instanceof_r`	dynamic type checks
`monitorenter, monitorexit`	`monitorenter_r, monitorexit_r`	synchronization
`aload, astore, areturn`	`rload, rstore, rreturn`	move pointers on the stack
`anewarray, aaload, aastore,`	`rnewarray, raload, rastore,`	operations on arrays of pointers

In addition, there is an instruction called `restrict` that implements the Luna `@` operator. The following sections describe the implementation of these instructions in more detail.

4.2 Implementing remote pointers

At a low level, Luna's `@` operator is simple to implement: two move instructions move a local pointer and a permit together so that they form a single, 8 byte remote pointer. This requires no synchronization or memory allocation. Once formed, remote pointers are handled much like other Java 8 byte values, such as `long` and `double` types. However, Java allows `long` and `double` types to be loaded and stored in heap objects non-atomically, in 4 byte portions. If one thread is writing a new 8 byte value into a memory location, another thread concurrently reading the location may end up with 4 bytes of the old value mixed with 4 bytes of the new value. This behavior could cause security risks for remote pointers, so Luna uses 8 byte atomic operations to load and store remote pointers in the heap shared between threads (operations on a thread's private stack need not be atomic). Luckily, the x86 architecture supports several instructions that move 8 bytes atomically. Unluckily, these instructions are all awkward—Luna ends up moving remote pointers through the floating point registers to implement an atomic load or store.

Like J-Kernel capabilities, Luna remote pointers support *immediate revocation*: as soon as a call to a permit's `revoke` method completes, Luna guarantees that no other thread can use the permit to read or write data. Unfortunately, this guarantee is difficult to implement efficiently, because operations on remote pointers require both an access control check and an

actual data access. If these operations are not performed atomically, then revocation may not be immediate. Suppose thread A tries to use the remote pointer e@p, while thread B concurrently revokes p and then modifies e. The figures below show two incorrect interleavings of these operations. In Figure 4.1, thread B modifies e, and the new value in e is incorrectly leaked to thread A, which should not be able to see this value, since B revoked p before changing e. In Figure 4.2, thread A incorrectly causes a side effect that is visible to thread B after B has revoked A's access.

Thread A	Thread B
1. check permit p	
	2. revoke p
	3. modify e
4. read e	

Figure 4.1: Read after revoke

Thread A	Thread B
1. check permit p	
	2. revoke p
	3. modify e
4. modify e	
	5. read e

Figure 4.2: Modify after revoke

To avoid these race conditions, remote field and array operations are protected by critical sections. An operation first acquires a lock, then checks the permit to make sure access has not been revoked, then, if access is granted, performs the operation and releases the lock. If access is denied, the operation releases the lock and throws an exception. Currently, Luna uses a single system-wide lock for these critical sections (the same lock that is used to protect Marmot's memory allocation operations).

On a uniprocessor, specialized thread schedulers can implement critical sections without explicit run-time locks [SCM99], but on a multi-processor, I am unaware of any way to avoid the race condition above without explicit locks or explicit coordination between processors. Anyway, Luna is

implemented over NT kernel threads and cannot modify NT's scheduler. Luna employs one trick for remote pointer reads (but not writes) on a uniprocessor: it first performs the read, then checks the permit, without ever entering a critical section. Unfortunately, relaxed cache consistency prohibits this optimization on multiprocessors.

The locks and the access check make a remote pointer operation more expensive than a local pointer operation. There is one additional cost due to lost optimization opportunities. Like Java, Luna defines the semantics of a program in a strict, deterministic way (except for multithreading, which introduces limited nondeterminism). Consider the following code:

```
List~ x = ...;
List~ y = ...;
x = x.next;
y = y.next;
```

Figure 4.3: Deterministic evaluation

Because the evaluation of the `x.next` and `y.next` expressions may throw revocation exceptions, Luna's code generator must execute the first expression before the second to preserve the semantics of the original program. This limits the optimizer's power to reorder code. In fact, ordinary Java suffers from the same problem—even if `x` and `y` are local pointers, it is difficult to reorder the expressions because the evaluation may throw null pointer exceptions. On the other hand, Java does relax the consistency requirements on data shared between threads; a thread can cache a "working copy" of shared data and read from this working copy rather than the actual shared data in some circumstances (the details of Java's consistency model are under debate; see [Pug99]). Luna could also be extended with this ability, thus

avoiding the need to perform revocation checks on data cached in the "working copy." The current implementation exploits this idea for cast operations on remote pointers, but not for field and array operations.

4.2.1 Special Java pointer operations: equality, instanceof, checkcast, synchronization

Java supports several special operations on pointers that do not fit neatly into Luna's local/remote pointer model. Java programs may query pointers for equality (with the `==` and `!=` operators) and test the type of a pointer with casts and `instanceof` operations. Remote pointers must support these operations, because pointer equality is used to test the end of a data structure, such as a list, and casts are used to overcome Java's weak static type system. It is not clear, though, what these operations should do when applied to a revoked remote pointer. On one hand, such an operation on a revoked remote pointer for equality could raise a revocation exception. Unfortunately, checking for revocation in these cases slows down program execution without providing any security benefit. Permits protect the data that a pointer points to—they do not necessarily need to protect the value and type of the pointer itself; the J-Kernel allowed equality and type tests on a revoked capability, and this was never a cause for concern.

On the other hand, omitting the revocation check is actually more difficult to implement than performing the revocation check, because the object pointed to by a revoked pointer may be garbage collected, which throws away the type information needed to implement cast and `instanceof` at run-time.

Because neither solution is entirely satisfactory, the Luna semantics allow an implementation to perform revocation checks for pointer equality and type tests, but does not require them. The current Luna implementation never checks for revocation in equality tests, but does check for revocation in type tests, except where the compiler is able to eliminate the dynamic type tests statically.

4.2.2 Synchronization

Since Luna supports communication through shared data as well as RMI-based communication, Luna requires some form of inter-task coordination. For example, two tasks sharing a data buffer may want to share a lock to coordinate reads and writes to the buffer. Another example is data copying: when a task copies another task's data, it must lock the data to make sure that it receives a consistent copy.

Luna assumes the use of lock-based rather than lock-free synchronization, since Java's language and libraries use locking for concurrency control. The most natural way to extend Java's synchronization mechanisms to Luna is to allow Java `synchronized` statements to act on remote pointers as well as local pointers. For example, a function to copy a remote `Vector` object acquires a lock on the remote object:

```
public class Vector {
    ...
    public static Vector copy(Vector~ from) {
        synchronized(from) {
            Vector to = new Vector();
            ... copy data from “from” to “to”...
            return to;
        }
    }
}
```

Figure 4.4: Remote synchonization

This inter-task task synchronization mechanism raises several issues, only some of which Luna resolves satisfactorily. First, shared locks present a denial-of-service risk: by acquiring a lock and never releasing it, a task can prevent another task from making progress. This is particularly problematic for Java's `synchronized` statement, which can be applied to *any* object to which a task has access. This breaks an object's encapsulation, since part of its state (its locking status) is exposed to anyone who has a pointer to the object. To fix this, Luna only allows a class A to synchronize on remote pointers of class A (or of a subtype of A, since these can be coerced to type A). This protects synchronization in the same way that Java's `private` qualifier protects fields and methods, and enables the sharing of ADT's between tasks: if two classes share abstract datatype objects, such as `Vector`, they trust the shared ADT implementation to never hold locks indefinitely. Like the J-Kernel, Luna relies on trust to effectively share abstract datatypes.

Unfortunately, it is still possible to hold locks on local pointers indefinitely, under Java's permissive rules for local pointers, which means that a task must be wary of acquiring locks on a remote object—the object's owner might be holding the lock forever. Ideally, Luna would change the access rules for local pointer synchronization as well as for remote pointer synchronization, but this would break backwards compatibility with Java.

When a task is terminated, it may still hold locks on other task's objects. To prevent this from blocking other tasks' progress forever, Luna releases a task's remote locks when the task is stopped. Naturally, this raises the concerns about *damaged objects* that motivated the task model in the first place—if a task is shut down while it is atomically modifying a remote object,

the object is left in an inconsistent state. This problem is solved with a simple convention: tasks should never modify remote objects in any way that could leave them in a damaged state. Preferably, remote locks should be used only for reading, not for writing. The most common use of remote locks is to copy remote data, which satisfies this convention. If a task needs to update a remote object atomically, it must make a cross-task method invocation to the task that owns the object, so that the update is performed locally, protected by a local lock acquisition. In this way, cross-task method invocations provide a safe fallback when there's no shared data algorithm that is robust to task termination.

In fact, one could argue that remote locks are not necessary at all, because remote synchronization can be encoded with local synchronization and cross-task communication. The `Vector` example above can be written as:

```
public class Vector {
    ...
    public static Vector copy(Vector~ from) {
        Vector to = new Vector();
        Permit p = new Permit();
        from.copy1(to @ p);
        p.revoke();
        return to;
    }
    private synchronized void copy1(Vector~ to) {
        ... copy data from "this" to "to"...
    }
}
```

Figure 4.5: Synthesizing remote locks

Under this implementation, task A copies task B's `Vector` object by transferring control to B's `copy1` function, which acquires a lock and copies the data. Unfortunately, this raises the cost and complexity of copying. Furthermore, B cannot allocate any objects in A's task directly, so if `copy1`

needs to allocate objects in A it must perform a cross-task method invocation back to A, while still holding the lock in B. This implementation also undermines Java's nested lock acquisition rules: a single thread may reacquire a lock as many times as it wishes without deadlocking, but cross-task method invocations involve multiple threads. If A calls B, which acquires a lock and calls A, which calls B and tries to acquire the same lock again, it will deadlock. Because of these problems, Luna's remote synchronization is a sensible feature.

4.3 Method invocations and threads

A method invocation on a remote pointer calls another task's code, and therefore must execute in one of the other task's threads. This thread switch allows the caller thread to be killed without abruptly terminating the callee thread. However, Luna is implemented over Win32 kernel threads, and switching kernel threads is an expensive operation. Therefore, cross-task calls only switch kernel threads lazily, when a call must be interrupted. In the normal case, a cross-task call only switches stacks, which can be done without involving Win32. To see how this works, consider a method in task A which calls a method in task B, which in turn calls a method in task C. This sequence executes in a single kernel thread, but involves 3 separate stacks (see Figure 4.6). If task B is now terminated, B's stack is deallocated, C's method continues to run in the original kernel thread, and a new kernel thread is allocated which resumes A's method (and raises an exception in A's method to indicate that the call to B aborted abnormally). This model is similar to RPC models described for the Mach [FL94] and Spring [HK93] microkernels. It retains the advantages of kernel threads, such as scheduling that interacts properly with

blocking I/O, while eliminating the cost of switching kernel threads in the common case. Unlike the J-Kernel's thread segment mechanism, it allows immediate and full cleanup of a terminated task's stack space and heap data.

The details of a cross-task method invocation are as follows. First, the invocation enters a critical section. Then it checks the remote pointer's permit for revocation, and throws an exception if the permit is revoked. The permit also holds a pointer to the task that created the permit. Each task keeps a free list of available stacks, and the invocation retrieves a stack from this list (or allocates a new stack if the free list is empty).

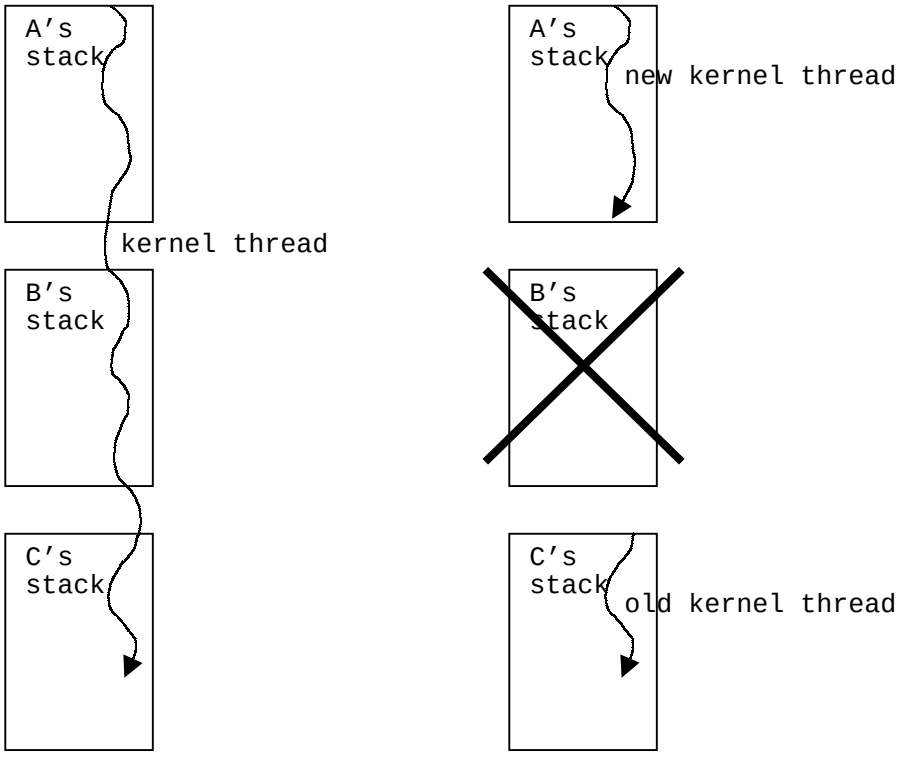

Figure 4.6: Threads and stacks in cross-task invocations

Next, the invocation performs some bookkeeping work. The bottom of each stack contains a block of bookkeeping data, and the top of the stack is aligned to a power of 2 boundary, so that the location of the bookkeeping data is calculated with a simple bit-wise && operation followed by an || operation. Currently, stacks are 64 kilobytes long, and a virtual memory guard page prevents execution from running off the top of the stack. A cross-task method invocation saves the following information in a stack's data area:

- a pointer to the kernel thread, to help the termination mechanism find and suspend a stack's thread while the stack is being destroyed,
- the permit of the remote pointer, used to implement cross-task exception handling, and
- a pointer to caller's stack, saved in the callee stack data, and a pointer to the callee's stack, saved in the caller stack data, so that a chain of stacks in a single kernel thread form a doubly linked list.

The doubly linked list is used during the termination process: in Figure 4.6, the termination mechanism must find A's stack and C's stack when task B is terminated. In retrospect, this explicit list may be superfluous—the callee's prologue saves the caller's frame pointer on the callee stack, which could be used as a link from the callee to the caller (assuming a safe point mechanism, described below, that can advance a thread past a prologue or epilogue), and the link in the other direction could be computed on demand by traversing the entire list of callee-to-caller links.

In addition, the invocation saves the current stack pointer in the caller's stack frame, at a standard offset relative to the frame pointer. This is used to

restore the stack pointer when the invocation returns or if an exception is thrown from the callee to the caller.

Finally, the invocation switches the stack pointer to point to the callee stack frame, exits the critical section, pushes the arguments to the call, and calls the callee code. The order of these operations is constrained slightly by on-demand termination. Luna must assume that both the caller and callee tasks may disappear at any moment, deallocating both the stacks and threads used by the call. Ideally, critical sections would protect the entire call process, including the transfer of control from the caller code to the callee code, from abrupt termination. However, this would require the callee code to exit the critical section. Unfortunately, the callee code is just an ordinary Java method, which does not know whether it is being called locally or remotely. Therefore, the caller must exit the critical section before calling the callee, and for a brief period of time, the thread runs the caller's code while using the callee's stack. If the caller or callee is terminated during this period, Luna's termination mechanism recognizes the situation by comparing the stack pointer (pointing to the callee stack) with the frame pointer (still pointing to the caller stack), and it aborts the cross-task call. This does not affect the program semantics, but it requires the implementation to deal with a tricky special case.

A normal return from a remote method invocation enters a critical section, restores the caller original stack pointer, returns the callee stack to its free pool, nullifies the caller's pointer to the callee, and exits the critical section. In total, a cross-task call requires entering and exiting a critical section twice, just as in the J-Kernel.

The callee may also return control to the caller by throwing an exception. Exceptions in Java are objects, so an exception thrown across a task

boundary must be referred to by a remote pointer. This remote pointer consists of a pointer to the exception object paired with the permit of the remote pointer used to make the cross-task method invocation. Rather than changing Java's try/catch mechanism to catch remote pointers directly, Luna boxes the remote exception pointer inside a local `TaskNestedException` object, which the program can catch and then extract the boxed remote pointer.

4.3.1 Terminating threads

Like the J-Kernel, Luna stops a task's threads when the task is terminated. Because Luna's RPC model dissociates stacks and kernel threads, Luna's thread termination is driven by *stack termination*—Luna finds and deallocates a task's stacks, and then rearranges kernel threads to match the remaining stacks in the system. Each task keeps a list of its stacks. For each stack in a dying task's list, Luna looks up the stack's kernel thread and advances this thread to a safe point, so that Luna can analyze the state of the thread. For example, being at a safe point ensures that a thread is not in the middle of executing a function prologue or epilogue, so that the frame pointer and stack pointer correctly describe the frames held by the kernel thread. In addition, safe points record garbage collection information, which is used to help clean up a terminated stack's state.

Consider the example above, where, in a single kernel thread, task A calls task B, which calls task C, and then task B is terminated. After advancing the kernel thread to a safe point, Luna sees that B's stack is blocked waiting for C's stack to return. Since C will have nothing to return to after B is gone, Luna

modifies the bottommost return address on C's stack, which used to point to B's code, to point to a function that quietly exits the thread when C returns.

Next, Luna cleans up any global state that the stack has established. Each stack keeps a list of the remote locks that it holds, and Luna traverses this list to free these locks. As described below in section 4.6.1, stacks may register their frames with a permit in order to receive immediate updates whenever a permit's state changes. Luna pops each stack frame one by one, using the garbage collection information in each frame to identify and unregister any frames registered with some permit.

Finally, Luna sees that A is blocked waiting for B to return. Since B will never return, Luna assigns a new kernel thread to run A's stack, and sets the initial instruction pointer of this kernel thread to an exception handler. The new kernel thread's frame pointer is established from the bottommost saved frame pointer on B's stack, and the thread's stack pointer is restored from its saved value in A's topmost frame. Callee-saves registers, however, are impossible to restore—it is conceivable that callee saves registers might be passed all the way from A through B to C before being spilled, and Luna's garbage collection information is not powerful enough to find the spilled values in C's stack frames. Therefore, A's exception handler must assume that all callee-saves registers are lost, and any values that the handler relies on must be spilled before making the cross-task invocation to B.

In the example above, B's stack was stopped while blocked on a cross-task method invocation. Luna must also be able to terminate threads that are blocked on kernel I/O, without leaving the I/O in a damaged state. The most obvious way to stop such thread is to use NT's `TerminateThread` function; however, NT's documentation warns that "if the target thread is executing

certain kernel32 calls when it is terminated, the kernel32 state for the thread's process could be inconsistent," so Luna uses a more cautious approach to stopping the blocked thread. Each Luna thread holds a special NT event that is signaled when the thread is stopped. Blocking I/O functions in Luna use asynchronous I/O functions, and then block waiting for either the I/O to complete or the thread's special event to be signaled, so that Luna can easily unblock the thread and cancel the asynchronous I/O (using NT's `CancelIo` function) when the thread is stopped.

4.4 Garbage collection and safe points

Luna extends Marmot's stop-and-copy garbage collector to deal with permits and to support safe points at backwards branches. When the collector traverses a remote pointer, it checks to see whether its permit was revoked. If the permit is unrevoked, then the remote pointer is treated as a strong pointer, while if the permit is null or revoked, the remote pointer is treated as a weak pointer, since the revocation makes the object semantically unreachable through the remote pointer. This allows a task's objects to be garbage collected even if there are outstanding revoked remote pointers to the objects. In particular, when a task is terminated, all of the task's permits are automatically revoked, causing all the task's objects to become collectable, so that a task's resources are reclaimed when the task is terminated.

Marmot's garbage collector cannot run until all threads reach a safe point, so that Marmot can examine each stack's state accurately. Unfortunately, Marmot only places safe points at allocation points and blocking system calls. Under this approach, a thread can run forever without allocating memory or blocking on a system call, so that it never reaches a safe

point, and the garbage collector can never run. Since Luna supports untrusted applications, it cannot allow a thread to perform this kind of denial of service attack. Therefore, Luna adds safe points at method calls and backwards branches. For each method, Luna keeps an array of safe point locations. To advance a thread to a safe point, Luna suspends the thread, copies the method's code into a temporary buffer, overwrites each safe point location in the temporary buffer with a breakpoint instruction, flushes the instruction cache, changes the thread's instruction pointer to point to the temporary buffer, resumes the thread until it encounters a breakpoint, and then transfers the instruction pointer back to the original code, advanced to the safe point location reached in the temporary buffer. By acting on a copy of the code rather than modifying the original code, the advancer allows other threads to concurrently run in the original code. This enables Luna to use safe points to terminate a task's threads, without suspending all the threads in the system.

Native code and exception handling complicate Luna's safe point mechanisms. Calls to native code must either poll to check if Luna needs to interrupt a thread, or, if the native code needs to block indefinitely, register a marker on the stack that Luna uses as a starting point when scanning the stack. Exceptions are tricky because they may be thrown at non-safe points in the code (since practically any pointer operation can throw a null pointer exception, it seemed wasteful to make every exception throwing point a safe point). This means that the exception handling process must not allocate any memory until the thread can be advanced to a safe point. Otherwise, this memory allocation could trigger garbage collection while the thread is still at a non-safe point, which would deadlock the exception handling process. Because of this, Luna's exception handling process first advances the thread to

a valid exception handler in the Java code, which is a safe point, then allocates the exception object and fills in the exception object's stack trace, and then runs the handler.

4.5 Tasks and dynamic loading

The J-Kernel, implemented on commercial machines with dynamic loading, had many class loading features already at its disposal. Luna, on the other hand, has to cobble together its dynamic loading mechanisms from scratch, based on a virtual machine (Marmot) that was designed to favor running times over compile times. Consequently, Luna's dynamic loading is still done partially by hand, although it would not be difficult to automate, based on techniques from the J-Kernel. Luna code for a task is compiled with guavac to produce bytecode, which then passes through the Marmot compiler to produce a collection of assembly language files. These are assembled to produce binary coff files. When a new task is launched, Luna's dynamic loading mechanisms load and link the coff binaries into a running Luna virtual machine. When the task is terminated, the binaries are immediately unloaded. The garbage collector does not need to run before the code is unloaded, because Luna's termination mechanisms ensure that the code is unreachable.

Marmot compiles whole programs at once, including every class that a program relies on. Every library class used by the program (even `java.lang.Object`) is recompiled whenever the program is recompiled. Originally, Luna followed this model, which meant that each task loaded every class separately. Tasks did not share the compiled code for classes. This quickly proved impractical, because Marmot's run-time system (and Luna's

extensions to it) are largely written in Java, and it is extremely unnerving to have multiple copies of the run-time system's own classes floating around, some of which are liable to be unloaded it any time. Furthermore, replicating the same code in each task wastes memory, potentially degrading locality.

On the other hand, giving each task its own classes gives each task its own static fields, solving one of the difficulties in the J-Kernel's linking mechanisms. Even better, it gives each task the opportunity to specialize the compiled code, based on static whole-program analysis of all the classes used by the task. Marmot performs several whole-program optimizations: inter-module inlining, uninvoked method elimination, static method binding, and stack allocation of objects. Luna preserves all of these except for stack allocation, which adds entries to method tables in an unpredictable way; Method table layouts must be consistent across all tasks for cross-task method invocations to work correctly, which means that integrating Marmot's stack allocation into Luna would require some effort.

Because there are both advantages and disadvantages to sharing code between tasks, Luna implements a compromise—code for some classes is shared, and code for other classes is not. Currently, the set of classes that share code, listed in Table 4.2, is hard-wired. Some of these classes, such as `String`, were hand-modified to implement per-task static state rather than global static state. A more flexible and automatic approach to sharing code would be desirable.

Table 4.2: Classes that share code (all are in java.lang)

Object, Class, VTable, Cloneable, Monitor, Thread, KThread, UThread, Task, ThreadPool, ListBase, Revocable, RevocableTree, Permit, RegisterFrame, Mother, String, StringBuffer, Character, Runnable, Throwable, Error, Exception, RuntimeException, StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException, ArrayStoreException, NullPointerException, InternalError, ArithmeticException, TaskException, IllegalMonitorStateException, TaskDeathException, TaskNestedException, RevokedException, NegativeArraySizeException, CloneNotSupportedException, IllegalThreadStateException, InterruptedException, IllegalArgumentException, VirtualMachineError

4.6 Micro-benchmarks and optimizations

Operations on remote pointers require synchronization, which is expensive on modern processors. On a 266MHz P6 processor, a lock followed by an unlock, both written in hand-optimized assembly using an atomic compare and exchange instruction, takes 80 cycles when measured in a tight loop. The bottleneck is access to the bus: on a multiprocessor, an atomic operation must "lock" the bus while it executes. On a uniprocessor, the bus lock may be safely omitted, which cuts the cost of a lock/unlock sequence to 20 cycles. Because the choice of uniprocessor vs. multiprocessor has such a large impact on the cost of remote pointer accesses, this section reports benchmark results on both uniprocessor and multiprocessor configurations.

Table 4.3 shows the performance of local and remote field accesses, and local and remote method invocations to a function with an empty body. Both the local and remote method invocations perform a simple method table dispatch. All measurements indicate the number of cycles taken on a 266MHz P6 processor with 64MB of RAM and 16K/512K L1/L2 data cache, measured in a tight loop (note: an empty loop takes 2 cycles per iteration; this was not subtracted from the numbers below).

Table 4.3: Remote pointer performance, without caching optimizations

	local	remote (uniprocessor)	remote (multiprocessor)
field read	3	5	94
field write	3	34	94
method invocation	10	111	228

The cross-task invocations, while slower than local invocations, are nevertheless faster than round-trip IPC on the fastest x86 uniprocessor microkernels [LES+97], faster then J-Kernel capability invocations, and orders of magnitude faster than Win32 LRPC calls. Unfortunately, the field accesses that require locks are slow to the point of being useless: if shared data were always so expensive to access, it would make more sense to copy data, as in the J-Kernel, than to share it. Luckily, standard *caching* and *invalidation* techniques apply to remote pointer accesses, because accesses to the data are more frequent than revocation of the data. This is analogous to virtual memory, which stores access control bits for each page in a page table. The cost of virtual memory's access control is acceptable because a translation lookaside buffer (TLB) caches the page table information, and is invalidated whenever the page tables are modified (which is expected to occur infrequently).

4.6.1 Caching permit information

This section describes how Luna imitates a hardware TLB to reduce the cost of repeated access checks of the same permit. However, Luna's approach differs from a hardware TLB in that a TLB operates entirely on dynamic information, while Luna takes advantage of static information to detect permit reuse. As a simple example, consider the following loop, which zeroes the elements of a remote list.

```
void zero(List~ list) {
    while(list != null) {
        list.i = 0;
        list = list.next;
    }
}
```

Figure 4.7: Permit caching example I

When translated into Marmot's typed high-level SSA representation, where each variable is read-only and initialized in one location, this loop looks like:

```
void zero(List~ list0) {
    for(;;) {
        List~ list1 = merge(list0, list2);
        if(list1 == null) break;
        list1.i = 0;
        List~ list2 = list1.next;
    }
}
```

Figure 4.8: Permit caching example II

As Luna compiles this function from a typed high-level representation to a typed low-level representation, it adds type information to indicate repeated uses of the same permit, similar to the typed representations described in section 3.4. This information is computed using a straightforward intraprocedural dataflow analysis on the SSA representation. The analysis assigns a *permit variable* to each variable x holding a remote pointer, according to the following rules:

- If variable x is initialized by reading a local pointer field, array element, or return value from a remote object held by variable y, then x inherits y's permit variable. The expression "`list2 =`

`list1.next`" falls under this case, since `next` is a local pointer field of the class `List`.

- If a variable x is initialized by merging together variables $y_1,...,y_n$, and $y_1,...,y_n$ all have the same permit variable, then x is also assigned this permit variable. The expression "`list1 = merge(list0, list2)`" falls under this case.
- Otherwise, a new permit variable is created for x. For example, `list0` is given a new permit variable.

For the code shown in Figure 4.8, one permit variable (call it ρ) is created and assigned to `list0`, `list1`, and `list2`:

```
Λρ.void zero(List{ρ} list0, permit{ρ} p0) {
    for(;;) {
        List{ρ} list1 = merge(list0, list2);
        if(list1 == null) break;
        list1.i = 0;
        List{ρ} list2 = list1.next;
    }
}
```

Figure 4.9: Permit caching example III

The intermediate representation shows that ρ is checked repeatedly in the inner loop. Based on this, Luna inserts code outside the loop to cache this permit's revocation flag in a register or stack slot, so that an access check is done with a simple test of the cached value, with no locking. At run-time, this caching code adds the current stack frame to a doubly linked list held by the permit, and removes the stack frame from the permit's list after the loop is finished. The cost to add and then remove a frame from a permit's list is about 60 cycles on a uniprocessor and 175 cycles on a multiprocessor (where a bus lock is needed to implement a critical section), which means that the

optimization pays off after about 2 or 3 accesses to the shared data. If the permit is revoked, Luna's run-time system invalidates the cached information in each stack frame in the list by suspending each affected frame's thread, advancing the thread to a safe point, and then using the GC information at the safe point to determine which registers and stack slots must be modified to invalidate the cached access control information.

The caching optimization removes locking from an inner loop, eliminating the major cost of Luna's revocation checks. In fact, in some circumstances it is possible to go further and completely eliminate the check in the inner loop. Luna performs this optimization under the following conditions: (i) the loop contains no call instructions, exception handlers, or other loops, (ii) the loop uses only one permit, and (iii) all paths inside the loop from the loop header block back to itself check the permit for revocation. If these are satisfied (the `zero` loop, for example, satisfies them), Luna places safe points before the loop's revocation checks rather than at the loop's backwards branches, and then omits the code for the revocation checks entirely. If the permit is revoked, and the thread is advanced to a safe point inside the loop, then the thread is left at a point where a revocation check would have appeared, and a revocation exception is raised asynchronously in the thread at this point, so that it appears to the user that there was actually a check in the code there. Since Luna advances Java code to a safe point by setting breakpoints rather than by polling, this optimization reduces the per-iteration cost of the revocation checks to zero.

These optimizations preserve the original semantics of the Luna program: they only throw exceptions at points where the original Luna program could have thrown an exception, and precisely reflect the state of the

program when the revocation exception is thrown. A call to `Permit.revoke` raises all the necessary exceptions before returning, so that afterwards, no threads can possibly access data using the permit (i.e. revocation is still immediate, not delayed). Stated more strongly, any possible program execution trace under these optimizations is also a legal execution trace in an unoptimized Luna implementation.

Figure 4.10 shows the assembly language code generated for the body of the `zero` loop, as well as the code generated for an equivalent loop over a local list. Except for a difference in the instruction used to test for null, they are identical, and both execute in 3 cycles per iteration. Nevertheless, while the inner loops are identical, the remote traversal must manipulate the permit's linked list before and after the inner loop (in a critical section, of course), and the cost of this is significant: Table 2.1 shows the cost of repeatedly zeroing the elements of the same list, for lists of size 10 and 100. Luna's static analysis makes the speed of these loops reasonable, but not as good as in the local case.

local list traversal

```
loop:
  /* list.i = 0 */
  mov dword ptr [eax+8],0
  /* list = list.next */
  mov eax,dword ptr [eax+12]
  /* if(list == 0) goto done */
  test eax,eax
  je done
  jmp loop
...
done:
```

remote list traversal

```
loop:
  /* list.i = 0 */
  mov dword ptr [eax+8],0
  /* list = list.next */
  mov eax,dword ptr [eax+12]
  /* if(list == 0) goto done */
  cmp eax,0
  je done
  jmp loop
...
done:
```

Figure 4.10: Inner loop assembly code for local and remote list traversals

Table 4.4: Speed of local and remote list traversals

	local	remote (uniprocessor)	remote (multiprocessor)
10 element list	46	105	221
100 element list	316	375	493

Even if Luna extended the static analysis to encompass more than inner loops, no amount of static analysis can eliminate all the run-time checking, and Table 4.3 and Table 4.4 show that the penalty for each operation that acquires a lock is steep. This suggests that additional optimizations, such as dynamic reuse detection (e.g., as keeping a hash table of recently used permits in each thread) may also be needed to avoid lock acquisitions.

4.7 Luna web server

This section describes a Luna port of the web portion of the web/telephony server described in chapter 2. The server implements Sun's servlet API [Java]. To preserve this API, the Luna version of the server includes wrapper classes which lazily copy remote object data into local objects when the servlet requests the data, so that a servlet need not concern itself with remote pointers directly. Table 4.5 below shows the time taken to dispatch a request to a servlet (which lives in a different task from the server) and process the response, for a servlet which returns a fixed sized message, not counting the time to transfer the data to or from the underlying sockets, and not counting the cost of spawning a thread to handle the request. Table 4.5 shows Luna's performance with and without the caching optimizations, and compares this to the performance of a modified server where both the server and servlet reside in the same task, so that no remote pointers are used.

The measurements show that Luna's caching optimizations pay off for large messages, especially for a multiprocessor.

Table 4.5: Servlet response speed

	Single task (local pointers only)	Multiple tasks, no caching		Multiple tasks, caching	
		Uni-processor	Multi-processor	Uni-processor	Multi-processor
10 byte response	730µs	730µs	760µs	730µs	730µs
100 byte response	770µs	800µs	830µs	770µs	770µs
1000 byte response	770µs	1100µs	1400µs	970µs	1000µs

CHAPTER FIVE:
ALTERNATE APPROACHES TO THE TASK MODEL

The task model implemented by Luna and the J-Kernel differs significantly from Java's thread group model, which creates an opportunity to explore the design space between the two approaches. This chapter dissects the task model by breaking it into smaller pieces and examining these pieces in isolation. It also makes connections to other implementations of the task model, such as Alta [BTS+00], KaffeOS [BH99, BHL00], DrScheme [FFK+99], and JavaSeal [VB99]. This chapter is more speculative than the J-Kernel and Luna chapters—it is a mixture of past work, related work, future work, as well as some ideas that may not work.

An implementation of the task model must provide answers to the following questions:

- Termination: what is shut down when a task is terminated—objects, code, threads, or some combination of these?
- Task boundaries vs. code boundaries: are common abstract datatypes, like `String` and `Vector`, loaded separately into each task that uses them (either through sharing or replication), or do these datatypes live in one special task, so that every method invocation on one of these objects crosses a task boundary?
- Enforcing the task model: is this the responsibility of the programmer or does the system guarantee the task model (i.e. is enforcement voluntary or mandatory)?
- Symmetric vs. asymmetric communication: does the same communication mechanism apply to a downcall (an applet

calling a browser) as to an upcall (a browser calling an applet), or does the communication mechanism depend on trust in one direction? Can mutually suspicious tasks communicate?

- Server-centric vs. client-centric design: does a central server task perform much work, or is most work pushed into client tasks?
- Sharing: shared memory vs. RPC communication—is one preferred over the other? Should both be supported? What types of data may be shared between tasks—primitive types, abstract datatypes, immutable data, functions and methods, remote pointers, or RPC stubs?
- Revocation: should it be provided at all, or for only some types of operations (e.g. cross-task calls, but not field/array accesses)? Should revocation be immediate or delayed?
- Resource accounting policy: which task gets charged for each object?
- Garbage collection: does one heap suffice for all the tasks in the system, or does each task need its own heap?

The length of this list makes it impractical to explore every possible permutation of these questions. Instead, the rest of this chapter discusses these topics one by one, using each topic to highlight lucrative points in the design space and to point out important pitfalls in various approaches.

5.1 Termination, task boundaries

"What is shut down when a task is terminated—objects, code, threads, or some combination of these?"

"Task boundaries vs. code boundaries: are common abstract datatypes, like `String` *and* `Vector`*, loaded separately into each task that uses them (either through sharing or replication), or do these datatypes live in one special task, so that every method invocation on one of these objects crosses a task boundary?"*

This section breaks the task model into smaller pieces. Rather than terminating threads, objects, and code together, what if only a subset of these resources were deallocated when a task is shut down?

5.1.1 Terminating threads

To start, consider terminating only threads when a task is shut down. Java's thread groups implement this strategy, which creates problems of damaged state and failure to unload malicious code. Terminating threads may leave two forms of damaged state. First, if a client calls a server and kills the thread while the server is manipulating an internal data structure, the server's data structure is left in an inconsistent or deadlocked state. Traditional operating systems solve this problem either by not sharing threads between a client and server (i.e. switching threads when the client calls the server), or by placing the thread in a special "kernel mode" as it runs in the server, and delaying termination of the thread until it exits kernel mode. The J-Kernel and Luna are examples of the first approach, which has the advantage of extending to mutually suspicious communication. KaffeOS is an example of the second approach, which has the advantage of being easier to implement efficiently. KaffeOS contains two modes, user and kernel, and a thread running in kernel mode cannot be killed. Standard Java neither has kernel modes nor switches threads, leaving it vulnerable to damaged server state.

The second form of damaged state is *damaged objects*: abstract datatypes that are left in an inconsistent or deadlocked state when the thread that is modifying the object is terminated. The J-Kernel and Luna solve this problem by terminating objects as well as threads when a task is terminated, so that damaged objects become inaccessible. Alta and KaffeOS restrict the types of objects that tasks can share, so that it is not so easy to share abstract datatypes in the first place; if two tasks do share an object that relies on synchronization, then it is up to the programmer to implement this synchronization in a way that is robust to termination.

The standard Java API defines two ways of stopping a thread, `stop` and `destroy` (the former was deprecated, the latter is still not implemented, as of Java 1.3). `Stop` releases a thread's locks when killing the thread, leaving objects in an inconsistent state, while `destroy` does not release a threads locks, and therefore leaves objects in a deadlocked state. A better approach would be to place a thread's locks in a special "damaged state" when the thread is killed, so that any subsequent attempts to acquire the locks raise an exception, rather than succeeding (as with `stop`), or deadlocking (as with `destroy`). In other words, if the system cannot make damaged objects inaccessible, like Luna and the J-Kernel do, at least it should signal an error whenever a damaged object is accessed.

5.1.2 Terminating code

The thread termination approach has two serious problems: "good" code may be unintentionally interrupted, leaving damaged state, and "bad" code may continue to run. In general, there is no guarantee that "bad" code will not run again even after the "bad threads" are gone. This suggests a

different termination mechanism: rather than stop a task's threads, simply unload the task's code, and raise an exception whenever a thread attempts to run the unloaded code. In this approach, a task is defined by its code, rather than its objects or threads.

One nice feature of this model is that damaged objects are dealt with neatly. Suppose all of an object's methods are defined by a single task. As long as the task stays alive, the methods will never be interrupted, and the object will never be damaged; this is very different from the thread-termination model where built-in classes like `String` and `Vector` may be interrupted when some arbitrary applet's thread is stopped. If the task that defined the methods is killed, all of the methods stop working at once, and the object raises an exception whenever it is invoked. The situation is more complicated when overriding is taken into account, since a single object may have methods defined by multiple tasks.

As far as I know, code-based termination would be easy to implement. To implement immediate termination, the system can overwrite a task's code with breakpoints or illegal instructions (and flush the instruction cache), or change the virtual memory protection to make the code inaccessible (and flush the TLB). Eventually, the garbage collector reclaims the code's space by invalidating any function pointers to the unloaded code.

One potential difficulty in this scheme is inlining. If task A inlines task B's code, then if A gets terminated, this inlined code gets unintentionally interrupted. Likewise, if B gets terminated, then the inlined code fails to stop. Solving this problem would mean tracking the ownership of inlined code inside individual functions, and possibly forgoing some optimization

opportunities when it is impractical to track code ownership at such a low level.

5.1.2.1 Code sharing

Cross-task inlining may seem unimportant at first, since task crossings in traditional systems are infrequent. However, code-based termination leads to a very different notion of task crossing than found in Luna or the J-Kernel. In all language-based systems that I am aware of, common datatype classes (e.g. `String` and `Vector` in Java) are shared among tasks. By contrast, in the code-based termination approach, `String` and `Vector` belong to a different task than, say, an applet would belong to. Every call to a method of `String` or `Vector` is a task crossing in this model, so if the applet inlines calls to `String`, care must be taken not to interrupt the inlined code when the applet is shut down. Interrupting a call to `Vector`, for example, could produce a damaged `Vector` object.

Is this the right model, though? Perhaps an applet's call to a method of `Vector` should be interrupted when the applet is stopped. The Luna/J-Kernel model would stop a call to `String` or `Vector`, as long is it was a task-local call, rather than a cross-task call. If an applet manages to send one of the methods in `String` or `Vector` into an infinite loop, then the Luna/J-Kernel model would break this loop, while the code-based termination model would not.

The code-based termination does not explicitly recognize task crossings. For example, when a compiler generates code for a method invocation, the compiler does not always know whether the call will cross into another task or not. One drawback of this is that there is no opportunity to

automatically insert extra actions, such as revocation checks, at the boundaries between tasks.

5.1.3 Terminating threads and code

The Luna/J-Kernel task model ensures that a task's threads only run its own code. Any system that enforces this property automatically terminates a task's code when it terminates a task's threads. This raises the possibility of defining a task as a collection of threads and code together, but letting objects float freely between tasks. The danger of this is that overridden methods in these objects will allow one task's threads to call another task's code. The next section describes restrictions on object sharing to prevent this.

5.2 Sharing

"Shared memory vs. RPC communication: is one better than the other? Should both be supported? What types of data may be shared between tasks—primitive types, abstract datatypes, immutable data, functions and methods, or RPC stubs?"

Alta, KaffeOS, Luna, and the J-Kernel all enforce boundaries between tasks by restricting the types of objects that tasks may share:

- the J-Kernel and Luna support shared RPC stubs, which provide mediated access to other tasks' code,
- Luna supports remote (mediated) pointers to arrays and fields, and
- Alta and KaffeOS support direct (unmediated) pointers to other tasks' data, but not other tasks' code.

None of these systems support unmediated access to other tasks' code. This property is enforced by various mechanisms. The J-Kernel's copy mechanisms reject classes that contain foreign code. Luna mediates all data

access with permits and remote pointers. Alta and KaffeOS disallow sharing of any datatypes that might contain foreign code in overridden methods. Unfortunately, Java is biased towards classes whose methods may be overridden. For example, Alta and KaffeOS disallow the sharing of an object containing a field of type `Object`, because this field might hold an object that overrides one of `Object`'s methods, such as `equals` or `hashCode`. A pure object-oriented language, where all methods may be overridden, would be even worse. Nevertheless, this unmediated sharing is much easier to implement efficiently than Luna's mediated sharing. However, KaffeOS and Alta do not have a prominent RPC mechanism like Luna and the J-Kernel do, which limits their ease of use. Is there a way to combine the sharing in these systems that is easy to implement and supports both shared data and RPC communication? I believe that there is—but not necessarily in Java. The following section sketches such an approach for a non-object-oriented type system.

5.2.1 Sharing in non-object-oriented languages

To see the impact of object-orientation on sharing, consider a non-object-oriented type system consisting of primitive types, tuples, records, unions, arrays, recursive types, universally quantified polymorphic types, existentially quantified types, and functions. I hypothesize that all of these types, except functions, satisfy Alta's and KaffeOS's criteria that they cannot be used to call foreign code. That is, if a type contains no functions, then an object of that type may be safely shared between tasks, without accidentally importing foreign code into a task. This means that a task's threads only run its own code. Even though object sharing compromises the task model

somewhat (it is no longer clear who "owns" an object), a task's code and threads are still well defined.

So far, this hypothetical system supports shared data but not RPC. To rectify this, simply add a new type, the *capability*, which is exactly like a function except that it performs special cross-task functionality when invoked, such as checking for revocation and switching threads, and it may be shared between tasks. Capabilities need not copy data, since data may be shared directly.

Unfortunately, Java does not have tuples, records, unions, simple recursive types, universally quantified polymorphic types, or existentially quantified types. Besides primitive types and arrays, Java has classes, which contain functions and are therefore not shareable under the rules above. Perhaps this solution would be more appropriate for a non-object-oriented language like ML, although some work might be required to support modules. ML has the additional advantage of encouraging immutable data, which eliminates many of the problems with inter-task locks encountered in Luna.

5.2.2 Revocation

"Should revocation be provided at all, or for only some types of operations (e.g. cross-task calls, but not field/array accesses)? Should revocation be immediate or delayed?"

The previous section presented a hypothetical system in which RPC stubs (capabilities) support revocation, but other datatypes are shared without supporting revocation. This design has clear implementation benefits, since experience with Luna shows that revocation for field and array accesses is harder to implement efficiently than revocation for functions. However, Luna

assumed immediate revocation, implemented with explicit run-time checks. Another possibility is to use the garbage collector to implement revocation. In this implementation, revocation is delayed until the next full garbage collection, at least for field and array accesses. Having immediate revocation for method invocations but delayed revocation for field and array accesses is not orthogonal, but might be a reasonable compromise in practice.

5.3 Enforcing the task model, symmetric vs. asymmetric communication

"Is enforcing the task model the responsibility of the programmer or does the system guarantee the task model (i.e. is enforcement voluntary or mandatory)?"
"Symmetric vs. asymmetric communication: does the same communication mechanism apply to a downcall (an applet calling a browser) as to an upcall (a browser calling an applet), or does the communication mechanism depend on trust in one direction? Can mutually suspicious tasks communicate?"

A task system supporting untrusted code must have some mandatory enforcement of its task model, because it cannot trust all the tasks in the system to voluntarily obey the task model's constraints. However, the termination mechanisms may depend on the voluntary behavior of other, more trusted tasks. In KaffeOS, a dying task's threads are stopped, but only if the threads are not currently running kernel code. Thus the kernel's cooperation is needed to ensure that the task is terminated; in this case, a call from the task to the kernel must return in a reasonable amount of time.

As another example, DrScheme does not implement special mechanisms to avoid damaged objects. However, damaged objects may be avoided voluntarily if tasks explicitly relinquish pointers into dying tasks

(DrScheme's custodians do not do this automatically). If tasks do not relinquish these pointers, they risk seeing damaged objects, and they risk invoking code in the dead task.

On one hand, voluntary mechanisms are simpler to implement than mandatory mechanisms. DrScheme is not encumbered by the J-Kernel's heavyweight capabilities or Luna's complex revocation implementation. On the other hand, with extra support machinery, programmers can afford to be more daring, and share objects with fewer risks, because the system will take care of them. For example, Luna's remote pointers pinpoint the places where a programmer needs to manage sharing, and the mandatory revocation mechanisms ensure that mistakes cannot lead to a damaged object being accessed, or code in a dead task being run. This is similar to the way that static type safety and garbage collection allow a programmer to treat pointers far more freely than in a language like C or C++, despite protests from C programmers that type safety restricts pointer operations.

However, Luna's type system, like all decidable type systems, is conservative: it disallows some programs that could be proven safe, simply because the type system is not powerful enough to prove these programs' safety. An example of this is the way Luna forces programs to copy data even when it is clear that direct access to the remote data poses no dangers.

Another argument for mandatory mechanisms is that voluntary mechanisms often cannot guarantee certain properties. For example, from a language design standpoint it would be nice to guarantee that no damaged objects are visible to a program. Under a voluntary approach, a malicious applet could deliberately create a damaged object (e.g. by creating and then abruptly terminating a task), and pass this damaged object to the browser

through a method invocation. Even though the browser's code may respect the task model, it may still see damaged objects created by tasks that violate the task model. Similarly, compilers and run-time systems for safe languages cannot rely on programmer discretion for their safety. Luna's whole-task optimizations are only safe if the task model is enforced, which means that a system based on DrScheme's approach cannot perform these optimizations.

Even in a system that relies primarily on voluntary enforcement of the task model, extra mandatory mechanisms might guard against mistakes. For example, in DrScheme, overwriting a dead task's code with breakpoints would ensure that a parent does not accidentally call a dead child's code, even if the parent task is buggy.

5.3.1 Symmetric and asymmetric communication

When a user thread calls the kernel in KaffeOS, the kernel voluntarily changes the thread's mode. This voluntary mechanism requires the user thread to trust the kernel (a malicious kernel could take the thread and run forever, charging CPU time to the user thread). The kernel typically does not trust the user thread in the same way; nor does a user thread necessarily trust another user thread. Similarly, in DrScheme a child task trusts its parent task, but not vice-versa. This means that separate mechanisms are needed for parent-to-child communication or child-to-child communication, which makes communication less orthogonal than in the J-Kernel or Luna, which use the same mechanisms for child-to-parent, parent-to-child, and child-to-child communication. On the other hand, child-to-parent communication is more common than, say, sibling-to-sibling communication, so it may make sense to optimize the former and disregard the latter, both in terms of speed and ease-

of-use. In KaffeOS, this trade-off is quite explicit: downcalls into the kernel are fast and simple, while other communication proceeds through shared memory buffers, which are not as easy to work with as downcalls.

5.3.2 Server-centric vs. client-centric design

"Does a central server task perform much work, or is most work pushed into client tasks?"

Since the communication mechanisms in Luna and the J-Kernel assume mutual suspicion, task boundaries typically coincide with trust boundaries. For example, tasks are used to account for resource usage, and under mutual suspicion, task A is unwilling to give task B the right to charge resource usage to A's account (and vice-versa). In this case, resource usage boundaries coincide with trust boundaries. However, if A does trust B, it might allow B's code to run under A's account, so that B performs a service for A and this service is automatically funded. For example, traditional operating systems try to charge time spent in the kernel to specific user processes (which presumably trust the kernel). In a system where user processes are billed for resource usage, this maximizes billable time.

There are several plausible ways to realize this goal in a language-based task system. The first is to keep tasks aligned with trust boundaries, but to use threads to account for resources, rather than tasks, and to let threads cross task boundaries. This accounts for processor time well enough, but memory accounting is more problematic, since memory allocated by a thread can outlive the thread itself. A key goal of the task model is to eliminate this problem by ensuring that neither a task's memory nor a task's threads outlives the task.

A second idea is to align tasks with resource boundaries rather than trust boundaries. If a server performs a service for a client, and wants to charge the cost of this service to the client, then the server code should be loaded into the client task, so that it naturally runs on the client's account. Language safety protects the kernel's data from the client, even though they are both in the same task. KaffeOS and Alta encourage this design, since their emphasis on shared memory communication over RPC communication makes it easier for the client to call server code in the client task than to call server code in a separate task. Because the server runs in the client's task, it risks being stopped abruptly if the client task is terminated, possibly leaving system-wide shared server data structures in a damaged state. To prevent this, the server code might run in special mode (like KaffeOS's kernel mode) that postpones termination while the server code executes. However, the server code must be careful to always exit this mode before calling client code, so that the client code does not gain immunity against termination. This is subtle in an object-oriented language, since client code may reside in overridden methods of common classes; it is easy to accidentally call one of these methods (remember the security problems that forced Java's designers to disallow overridden methods in `String` objects). A key goal of the task model, at least as developed by Luna and the J-Kernel, is to explicitly mark the places where a server might end up calling client code, and to automatically switch threads when the server does call client code.

Looking at the two ideas above, it seems that when resource and trust boundaries do not coincide, there is a trade-off between using tasks to enforce trust boundaries vs. using tasks to enforce resource boundaries. Tasks align either with trust boundaries or with resource boundaries, but not both.

Perhaps this is only a result of stinginess, though—a third idea is to introduce more tasks, so that there are enough task boundaries to align with both trust and resource boundaries. Figure 5.1 shows the idea. The left side is a picture of client and server tasks in the J-Kernel or Luna, where resource, trust, and task boundaries all align. The right side splits each client into two tasks, one running the client code and the other running server code on behalf of the client, where both tasks' resources are charged to the client's account. A central server task manages activities that cannot be charged to a particular client. To traditional operating systems designers, this may seem rather baroque. After all, extra boundaries create extra costs, both in terms of performance and ease of use. However, language-based protection makes boundary crossings easier and cheaper, so that such baroque mechanisms are more practical. Whether this design works well in practice remains to be seen.

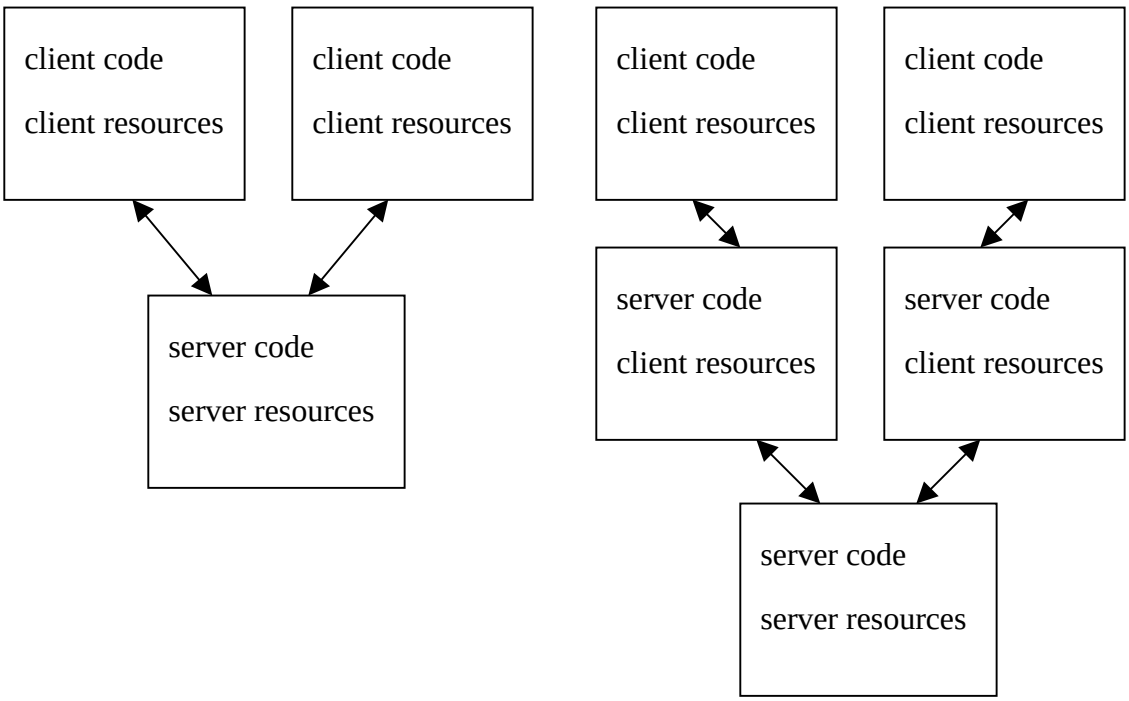

Figure 5.1: Expressing trust and resources with extra tasks

5.4 Resource accounting policy

"Which task gets charged for each object?"

Luna and the J-Kernel charge a task for the memory that it allocates. This works largely because a task's objects are deallocated when the task is terminated, ensuring that memory is never charged to a dead task. It has the advantage that it is easy for a task to predict for which memory it is being charged.

Another obvious policy is to charge a task for objects that are reachable from the task. The introduction to this thesis argued that this approach is dangerous when tasks share abstract datatypes, which make it very difficult for a task to predict for which memory it is being charged. However, there may be ways to eliminate this uncertainty. For example, KaffeOS tasks share data in special shared heaps, which are fixed in size and cannot hold pointers into other heaps, making it very easy for a task to predict what it will be charged for when using the shared heap. Tasks communicate through side effects on shared heaps.

For some languages, such as functional languages, fixed sized buffers and side effects are not appropriate. Perhaps tasks can share abstract datatypes, but the memory for the implementation of an ADT is charged to the task that allocated the ADT. As long as the ADT is deallocated when its allocating task is terminated, this is acceptable. However, this property is not necessarily easy to achieve. Consider the hypothetical type system from section 5.2, which supported shared abstractions through a combination of quantified types and capabilities (but not functions). If task B creates a capability, packs the capability in an object of type

$\exists\beta.(\beta * (int * int * int) * capability(\beta \rightarrow int))$

and passes this object to task A, then ideally A should be charged for the two tuples in the type, but should not be charged for the implementation of β or the implementation of the capability (the notation "capability(β→int)" indicates a capability that takes a β as an argument and returns an int). Intuitively, if task B is terminated, then the capability created by B is revoked, making β useless to task A, so that task A does not care if the implementations of the capability and of β are deallocated. It is not at all clear how to formalize this intuition, however. For example, suppose in the above example task B creates the capability, and another task, C, implements β and packs this together with B's capability in an object of existential type, which it then gives to task A. In this case, it is not obvious which task to charge for the implementation of β—charging task B is not fair, because C allocated the implementation of β, possibly without B's knowledge. Charging C is problematic because it might be terminated, leaving no one to charge for the implementation of β, which cannot be garbage collected because A and B are keeping it live. Task A could pay, but this violates the original goal: to make sure that A does not pay for someone else's abstract datatype.

5.5 Garbage collection

"Does one heap suffice for all the tasks in the system, or does each task need its own heap?"

While Luna and the J-Kernel implement a model in which it is clear which task to charge for which resource usage, they do not actually implement resource accounting and enforce resource limits. Some other projects have addressed this issue: JRes [CvE98], which was ported to the J-Kernel, limits a task's CPU, network, and memory usage, and KaffeOS also

supports similar limits. Enforcing CPU limits is not too problematic, as long as the enforcement is not too disruptive. For example, a thread shared between a client and server should not be stopped abruptly if the client runs out of memory, because this could leave the server in a damaged state. The task model, as implemented by Luna, the J-Kernel, Alta, KaffeOS, and DrScheme, ensures that no such damaged state arises.

However, enforcing memory limits is not as easy as enforcing CPU limits, even under a model with a clear resource accounting policy. Consider a system with 10 tasks, each having a 10MB memory limit. Suppose that 9 of the 10 tasks allocate memory very slowly, while one task rapidly allocates and throws away memory (perhaps it was written in ML). Despite its rapid allocation rate, this one task never has more than 1MB of live data, far below its 10MB limit. Suppose that the task has just allocated 10MB of data, only 1MB of which is still live, and the system detects that the task has reached the 10MB limit. It would not be fair to terminate or suspend the task at this point, because its live data is still far below the limit. Unfortunately, a system that allows data sharing between tasks (such as Luna or KaffeOS) cannot tell how much of the allocated data is live without running garbage collection. But it would be inefficient to garbage collect all 100MB in the system every time this one task allocates 10MB—one task with a very small limit could force the whole system to spend most of its time garbage collecting. The difficult problem is that one task is causing frequent collections, while other tasks are causing large collections, so that the system-wide product of the frequency of collections and the size of collections is much larger than the sum of the products of the frequency and size of the each task's individual contribution to garbage collection. In other words, the large time spent in collection is due

to an unfortunate interaction between different tasks' memory usage, rather than being the fault of any single task.

One solution is to allow tasks to overrun their limits arbitrarily in between garbage collections. After each collection, the live memory used by each task is computed and any tasks over their limits are suspended or terminated. One drawback of this is that it only catches a task long after it has exceeded the limits. However, under the J-Kernel and Luna's task model, this does not cause any permanent damage, because no matter how far the task went over its limit, all of its memory can be deallocated immediately by terminating the task, assuming that the system's policy is to terminate such a task immediately.

5.5.1 Accounting for collection time

A more subtle issue, raised by Back et al [BTS+00], is deciding which task should be charged for the cost (in CPU cycles) of the garbage collection process itself. The cost of a collection is proportional to the amount of live data in the system for a copying collector, and to the amount of live plus dead data for a mark-sweep collector, so perhaps each task should be charged for its live or live and dead data in each collection. However, this is not necessarily fair—suppose one task allocates a large amount data once, and keeps this memory live but never allocates any further memory. Ideally, it should be charged nothing for garbage collection, since it is doing nothing to cause garbage collection, but it will repeatedly be charged for its live data, depending on how fast the other tasks in the system allocate data.

An alternative is to charge a task in proportion to the allocation that it performs during each garbage collection cycle. If we assume an upper bound

on the fraction of the heap (or semispace) that is live at each collection, and we assume that allocation proceeds until the heap (or semispace) is completely full (i.e. we ignore fragmentation), then the GC cost charged to a task per byte allocated can be bounded statically by a constant, at least under simple models of stop-the-world collection. If this constant is reasonably small, then this solution is arguably fair—a task pays no more than a constant garbage collection tax on each allocated byte, regardless of what other tasks do. Whether these assumptions are reasonable, though, is another matter.

Another solution, implemented by KaffeOS, is to give each task its own heap, and to use distributed garbage collection techniques to garbage collect each heap separately. This strictly limits the amount of memory that a task can allocate, because it cannot overrun its own heap. For non-compacting collectors, it prevents fragmentation attacks; a task can fragment its own heap but cannot fragment any other task's heap. However, KaffeOS incurs a run-time cost when writing a pointer to the heap, because the distributed garbage algorithm uses a write barrier to detect cross-task pointers (although the static distinction between local and remote pointers in Luna and the J-Kernel might be able to eliminate many of these write barriers). In addition, the distributed garbage collection techniques have difficulties dealing with cycles between heaps, which KaffeOS solves in part by establishing special shared heaps between tasks, and allowing pointers into shared heaps but not out of shared heaps.

5.6 Conclusions

In the past few years, several interesting implementations of the task model have emerged. Yet, no single design seems superior in every respect.

Instead, a comparison between various approaches reveals tradeoffs between ease of use, ease of implementation, performance, flexibility, and guarantees of behavior. For example, mandatory enforcement of the task model is more difficult to implement than voluntary enforcement, but offers more flexible communication patterns and better guarantees of behavior. Similarly, allowing direct sharing of objects increases performance and ease of use, but complicates resource accounting. In light of these tradeoffs, the right approach depends on the programming language and the needs of the applications. Languages that frequently mix code and data, such as Java, benefit from rigid task boundaries to ensure that code does not migrate from one task to another, while less object-oriented languages may allow data to flow more freely across boundaries. Applications that load small, limited extensions may prefer the ease of implementation provided by voluntary enforcement, while environments with complex interactions between untrusted tasks require the guaranteed behavior that mandatory enforcement provides.

CHAPTER SIX:
CONCLUSIONS

In this thesis, I have argued that safe languages must incorporate operating system features in order to adequately support systems that execute untrusted code, such as browsers, servers, agent systems, databases, and active networks. Moreover, I have argued that adding small features in isolation does not adequately meet the needs of these applications. Instead, safe languages need large-scale aggregate structures that encapsulate each program's resources and define clear boundaries between different programs' code.

The two systems presented in this thesis, the J-Kernel and Luna, implement an aggregate structure called a *task*, in order to control a program's code, threads, and objects together. Thanks to this aggregation, tasks make guarantees that are impossible if these resources are managed separately. For example, terminating a program's threads without terminating its objects cannot guarantee the absence of damaged objects, and terminating a program's threads without terminating its code cannot guarantee that malicious code is unloaded on demand.

These concerns are especially relevant to an imperative object-oriented language like Java. In Java, abstract datatypes are protected by locks, raising the possibility of damaged objects, and malicious code may hide in overridden methods, making it difficult to eliminate all traces of a malicious program. Standard Java APIs for communicating with untrusted code address these issues by minimizing object orientation. These APIs typically accept non-

object-oriented data from untrusted code, such as `Strings` (which are final) and arrays, rather than non-final classes like `Vector` and `Hashtable`. This is analogous to C and C++ programmers who minimize dynamic memory allocation in order to avoid dangling pointers and memory leaks. By contrast, the J-Kernel and Luna provide mechanisms to automatically track the boundaries between tasks, giving programmers tools to manage communication between tasks with fewer risks.

The task model is not free; the J-Kernel and Luna sacrifice some performance to enforce task boundaries. First, control over threads requires extra work to manage calls from one task to another. Second, shutting down a task's objects requires support for revocation, which is difficult to implement efficiently for field and array accesses. Third, keeping "foreign code" from migrating into a task requires either restricting the types of objects that tasks can share, or copying objects to slice away or reject outside code. Our experience using the J-Kernel and Luna to build applications demonstrate that these costs are small compared to other Java overheads present in the applications. This suggests that an important area of research for extensible internet applications is how to design, implement, and compile these applications for maximum performance. I/O performance, in particular, is critical for web servers, active networks, and agent systems, and Luna's fine-grained sharing and revocation could improve this by exposing low-level hardware resources, such as network buffers, to applications.

Above all, the J-Kernel and Luna show that traditional operating system functionality does not require a traditional virtual-memory based implementation, nor does it require replacing well-typed language-based communication mechanisms between tasks with untyped traditional

operating system communication mechanisms. The J-Kernel's fast copy mechanisms demonstrate how strong typing and close cooperation between tasks yields orders of magnitude higher performance than loosely coupled mechanisms such as Java's RMI implementation built over serialization. Luna's fast cross-task method invocation implementation and special revocation optimizations show how cooperation between the compiler and run-time system can outperform traditional hardware-based implementations of communication and protection. Furthermore, the communication and protection mechanisms implemented by Luna and the J-Kernel provide a much higher level of abstraction to the application programmer than traditional operating systems or micro-kernels, and a stronger, more static enforcement of these abstractions.

There is an ongoing debate as to whether operating system designers should be abstraction merchants or performance fanatics. Language-based protection offers a way to implement operating system functionality that achieves both goals.

BIBLIOGRAPHY

[BCF+99] M. G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. *The Jalapeño dynamic optimizing compiler for Java.* ACM Conference on Java Grande, p. 129-141, Palo Alto, CA, June 1999.

[BH99] G. Back and W. Hsieh. *Drawing the Red Line in Java*. Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems, Rio Rico, AZ, March 1999.

[BHL00] G. Back, W. C. Hsieh, and J. Lepreau. *Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java*. Utah Technical Report UUCS-00-010, April 2000.

[BK99] Z. Budimli• and K. Kennedy. *Prospects for Scientific Computing in Polymorphic, Object-Oriented Style*. SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, TX, March 1999

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System*. 15th ACM Symposium on Operating Systems Principles, p. 267–284, Copper Mountain, CO, December 1995.

[BTS+00] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, J. Lepreau. *Techniques for the Design of Java Operating Systems.* To appear in Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, CA, June 2000.

[BV99] B. Bokowski and J. Vitek. *Confined Types*. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), Denver, Colorado, November 1999.

[CKD94] N. P. Carter, S. W. Keckler, and W. J. Dally. *Hardware Support for Fast Capability-based Addressing*. Sixth Int'l Conference on Architectural Support for Programming Languages and Operating Systems, p. 319–327, San Jose, CA, 1994.

[CV65] F. J. Corbató and V. A. Vyssotsky. *Introduction and Overview of the Multics System*. AFIPS Conference Proceedings (1965 Fall Joint Computer Conference), p. 185-196, 1965.

[CvE98] G. Czajkowski and T. von Eicken. *JRes: A Resource Accounting Interface for Java.* ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98), Vancouver, BC, October 1998.

[CWM99] K. Crary, D. Walker, and G. Morrisett. *Typed Memory Management in a Calculus of Capabilities*. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

[DDG+96] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. *Vortex: An Optimizing Compiler for Object-Oriented Languages*. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96), San Jose, CA, October 1996.

[DP93] P. Druschel and L. L. Peterson. *Fbufs: A high-bandwidth cross-domain transfer facility*. Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, p. 189-202, Dec. 1993.

[EB79] K. Ekanadham and A. J. Bernstein. *Conditional Capabilities*. IEEE Transactions on Software Engineering, p. 458–464, September 1979.

[FFK+99] M. Flatt, R. B. Findler, S. Krishnamurthi and M. Felleisen. *Programming Languages as Operating Systems (or, Revenge of the Son of the Lisp Machine)*. International Conference on Functional Programming (ICFP), Paris, France, Sep. 1999.

[FKR+99] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgard, D. Tarditi. Marmot: An Optimizing Compiler for Java. Microsoft Research Technical Report MSR-TR-99-33, June 1999.

[FL94] B. Ford and J. Lepreau. *Evolving Mach 3.0 to a Migrating Thread Model*. Proceedings of the Winter Usenix Conference, January 1994.

[GKC+98] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. *D'Agents: Security in a multiple-language, mobile-agent system*. Mobile Agents and Security, volume 1419 of Lecture Notes in Computer Science, p. 154-187, Springer-Verlag, 1998.

[GMS+98] M.Godfrey, T.Mayr, P.Seshadri, and T. von Eicken. *Secure and Portable Database Extensibility*. Proceedings of the 1998 ACM-

SIGMOD Conference on the Management of Data, p. 390-401, Seattle, WA, June 1998.

[HCC+98] C. Hawblitzel, C. C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. *Implementing Multiple Protection Domains in Java*. 1998 USENIX Annual Technical Conference, p. 259-270, New Orleans, LA, June 1998.

[HCU92] U. Hölzle, C. Chambers, and D. Ungar. *Debugging Optimized Code with Dynamic Deoptimization*. ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 32-43, San Francisco, June, 1992.

[HEV+98] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. *Implementation and Performance of the Mungi Single-Address-Space Operating System*. Software: Practice & Experience, p. 901-928 , July 1998.

[HK93] G. Hamilton and P. Kougiouris. *The Spring Nucleus: a Microkernel for objects*. Proceedings of the Summer 1993 USENIX Conference, p. 147-159, Cincinnati, OH, June 1993.

[HLP98] R. Harper, P. Lee, and F. Pfenning. *The Fox Project: Advanced Language Technology for Extensible Systems*. Technical Report CMU-CS-98-107, Carnegie Mellon University.

[HvE99] C. Hawblitzel and T. von Eicken. *Type System Support for Dynamic Revocation*. ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA, May 1999.

[IKY+99] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Kamatsu, and T. Nakatani . *Design, implementation, and evaluation of optimizations in a just-in-time compiler*. ACM Conference on Java Grande, p. 119-128, Palo Alto, CA, June 1999.

[Java] JavaSoft. *Java Servlet API*. http://java.sun.com.

[Javb] JavaSoft. *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?* JDK 1.2 API documentation, http://java.sun.com.

[Javc] JavaSoft. *The Java Hotspot Performance Engine Architecture*. http://java.sun.com/products/hotspot/whitepaper.html.

[Javd] JavaSoft. *Remote Method Invocation Specification*. http://java.sun.com.

[JL78] A. K. Jones and B. H. Liskov. *A Language Extension for Expressing Constraints on Data Access*. Communications of the ACM, Volume 21, Number 5, p. 358–367, May 1978.

[LES+97] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, T. Jaeger. *Achieved IPC Performance.* 6th Workshop on Hot Topics in Operating Systems, Chatham, MA, May 1997.

[Lev84] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[ML97] A. Myers and B. Liskov. *A Decentralized Model for Information Flow Control*. 16th ACM Symposium on Operating System Principles, p. 129–142, Saint Malo, France, October 1997.

[MWC+98] G. Morrisett, D. Walker, K. Crary, and N. Glew. *From System F to Typed Assembly Language*. 25th ACM Symposium on Principles of Programming Languages. San Diego, CA, January 1998.

[MS99] G. Muller and U. Schultz. *Harissa: A hybrid approach to Java execution*. IEEE Software, p. 44-51, March 1999.

[NL98] G. Necula and P. Lee. *The Design and Implementation of a Certifying Compiler*. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation. Montreal, Canada, June 1998.

[PCH+82] F. J. Pollack, G. W. Cox, D. W. Hammerstrom, K. C. Kahn, K. K. Lai, and J. R. Rattner. *Supporting Ada Memory Management in the iAPX-432*. Symposium on Architectural Support for Programming Languages and Operating Systems, p. 117-131, Palo Alto, CA, 1982.

[PDZ99] V. S. Pai, P. Druschel, W. Zwaenepoel. *IO-Lite: A Unified I/O Buffering and Caching System*. 3rd Symposium on Operating Systems Design and Implementation, p. 15-28, New Orleans, LA, February 1999.

[Pug99] W. Pugh. *Fixing the Java memory model.* ACM Conference on Java Grande, p. 89-98, Palo Alto, CA, June 1999.

[Ras86] R. Rashid. *Threads of a New System*. Unix Review, p. 37–49, August 1986.

[Red74] D. D. Redell. *Naming and Protection in Extendible Operating Systems*. Technical Report 140, Project MAC, MIT 1974.

[Sar97] V. Saraswat. *Java is not type-safe*. http://www.research.att.com/~vj/bug.html, August 1997.

[SCH+98] D. Spoonhower, G. Czajkowski, C. Hawblitzel, C. C. Chang, D. Hu and T. von Eicken. *Design and Evaluation of an Extensible Web & Telephony Server based on the J-Kernel*. Cornell Technical Report TR98-1715, 1998.

[SCM99] O. Shivers, J. W. Clark and R. McGrath. *Atomic heap transactions and fine-grain interrupts*. Proceedings of the 1999 ACM International Conference on Functional Programming (ICFP), Sep. 1999, Paris, France.

[Sha97] Z. Shao. *Typed Common Intermediate Format*. 1997 USENIX Conference on Domain-Specific Languages, Santa Barbara, California, Oct. 1997.

[Sma97] C. Small. *MiSFIT: A Tool For Constructing Safe Extensible C++ Systems*. Third USENIX Conference on Object-Oriented Technologies, June 1997.

[Spa89] E.H. Spafford. *The Internet Worm: Crisis and Aftermath*. Communications of the ACM, p. 678-687, June 1989

[SS75] J. H. Saltzer and M. Schroeder. *The Protection of Information in Computer System*. Proceedings of the IEEE, Volume 63, Number 9, p. 1278–1308, September 1975.

[SSF99] J.S. Shapiro, J.M. Smith, D.J. Farber. *EROS: a fast capability system*. 17th ACM Symposium on Operating System Principles, p. 170–185, Kiawah Island, SC, December 1999.

[TT94] M. Tofte and J.P. Talpin. *Implementation of the Typed Call-by-Value Lambda Calculus using a Stack of Regions*. 21st ACM Symposium on Principles of Programming Languages, p. 188–201, Portland, OR, January 1994.

[TMvR86] A.S. Tanenbaum, S.J. Mullender, and R. van Renesse: *Using Sparse Capabilities in a Distributed Operating System*. 6th

International Conference on Distributed Computing Systems, IEEE, p. 558-563, Cambridge, Massachusetts, May 1986.

[vdLin] P. van der Linden. *Java Programmers FAQ*. http://www.afu.com/javafaq.html.

[VB99] J. Vitek and C. Bryce. *The JavaSeal mobile agent kernel*. In First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA'99), October 1999.

[WA99] D. C. Wang and A. W. Appel. *Safe Garbage Collection = Regions + Intensional Type Analysis*. Princeton Technical Report TR-609-99, 1999.

[WBD+97] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. *Extensible Security Architectures for Java.* 16th ACM Symposium on Operating Systems Principles, p. 116–128, Saint-Malo, France, October 1997.

[WdFD+98] M. Weiss, F. de Ferriere, B. Delsart, C. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Siebert, and X. Spengler. *TurboJ, a java bytecode-to-native compiler*. LCTES'98, LNCS 1474, Montreal, Canada, June 1998.

[WLA+93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. *Efficient Software-Based Fault Isolation*. 14th ACM Symposium on Operating Systems Principles, p. 203–216, Asheville, NC, December 1993.

[WLH81] W. A. Wulf, R. Levin, and S. P. Harbsion. *Hydra/C.mmp:An Experimental Computer System*. McGraw-Hill, 1981.

[Wet99] D. Wetherall. *Active network vision and reality: lessons from a capsule-based system*. 17th ACM Symposium on Operating System Principles, p. 64–79, Kiawah Island, SC, December 1999.