Implementation Issues in Concurrent Programming Languages: A Framework for Thread Specialization

by

GREGORY DAVID BENSON

B.S. (University of California, Davis) 1991
M.S. (University of California, Davis) 1993

DISSERTATION
Submitted in partial satisfaction of the requirements for the degree of
DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

Professor Ronald A. Olsson (Chair)

Professor Carole M. McNamee

Professor Matthew K. Farrens

Committee in Charge

1999

Implementation Issues in Concurrent Programming Languages:
A Framework for Thread Specialization

The research was supported by the NSA University Research Program
grants MDA904-94-C-6109 and MDA904-96-1-0120.

Abstract

An increasing number of applications and system programs are being implemented as concurrent programs. In addition, the hardware required to support true concurrency is becoming more common. Fundamental to a concurrent program is the notion of a *thread* or *process*, which is an independent execution stream. Whereas a sequential program consists of a single thread, a concurrent program consists of two or more threads that work together to perform a task. In some cases, concurrency is used to gain better performance by taking advantage of multiprocessors and networks of computers, or by reducing hardware input/output (I/O) latency by having different threads ready to run in the presence of I/O delays. In other cases, concurrency is used because of the natural structure of a program. For example, many applications need to respond to asynchronous events such as user input or requests in a client/server system (e.g., web servers and file servers).

Programming languages with semantic support for concurrency — such as SR, Java, Modula-3, and Ada — can help manage the complexity of programming modern hardware. However, the increasing diversity in parallel and distributed platforms, both in hardware architectures and in operating systems, is creating a growing gap between language implementations and target platforms, making it increasingly difficult to develop efficient and portable compilers and run-time systems. Unlike sequential languages, concurrent languages rely on a run-time system to provide support for threads and possibly communication.

This dissertation examines this growing gap between concurrent languages and the underlying platforms. The question is: How should language-specific thread mechanisms be implemented in compilers and run-time systems for concurrent programming languages? Predominant approaches are either ad-hoc and specific to a particular language or run-time system, or use traditional software layering to handle the complexity of managing multiple targets but sacrifice performance.

The first contribution of this dissertation is to explore the implementation issues of a particular concurrent programming language, SR, on a diverse range of target environments, including microkernels, parallel run-time libraries, POSIX threads, and a bare hardware implementation.

The second contribution of this dissertation is to classify and analyze current techniques for implementing threads in concurrent run-time systems. This analysis will

help future language implementors understand the advantages and limitations of current approaches.

The third and most significant contribution of this dissertation is the *Mezcla thread framework*, which addresses the tension between portability and efficiency found in current thread systems. Many thread packages support only a limited number of target configurations and are generally inflexible with respect to scheduling. Even configurable thread packages distance core thread operations and thread state from client code. In addition, most thread implementations duplicate some of the functionality found in concurrent language run-time systems. The Mezcla thread framework addresses these limitations by enabling a language implementor to generate specialized, light-weight thread systems based on the requirements of the run-time system and the functionality of the target platform.

The approach taken in the Mezcla thread framework is based on three principles: eliminate redundancies found between run-time systems and thread packages; facilitate the best match between language mechanisms and the underlying platform; and allow run-time systems to be retargeted to different platforms and configurations.

This dissertation outlines detailed design issues for the Mezcla thread framework, describes a prototype implementation of the framework, and presents performance results. The Mezcla thread framework represents a step toward elevating thread system implementation from a simple library-based approach to the status of a compiler-based approach. The framework serves as model for future tools and operating systems.

To Laura

Contents

List of Figures

List of Tables

Acknowledgements

The research presented in this dissertation would not have been possible without the guidance, support, and dedication given by my advisor, Ron Olsson. Over the years he has influenced all aspects of my academic development. Most importantly, he has taught me how to write, present, and teach. I now have a career that I love. Thanks Ron!

Many other people have influenced my education and my success in academics. It is easiest to list people in chronological order. I thank Phil Windley for my first exposure to real computer science and the opportunity to work in the UC Davis Robotics research laboratory. Steve Hsia for his support of my work in robotics as an undergraduate. Ty Lasky and Bo Preising for giving me responsibilities and confidence. Jean Stratford for providing interesting work at the Institute of Governmental Affairs. Dick Walters for letting me take graduate courses in my senior year and for going to bat for me.

As a graduate student I was able to work with many stimulating people. I thank Karl Levitt for convincing me to do research in computer science. My quals committee, Matt Bishop, Matt Farrens, Karl Levitt, and Carole McNamee, for providing feedback on my early ideas. Greg Andrews for providing insightful comments on my thesis proposal and continued guidance.

In my later years as a graduate student, Koen Langendoen and Henri Bal enabled me to work as a visiting researcher at Vrije University in the Netherlands. Jay Lepreau gave me the opportunity to work with the Flux research group at the University of Utah. These positions helped me build a technical foundation from which I was able to carry out the research in this dissertation. I thank Matt Haines for many interesting conversations on thread systems and for buying the beer.

The completion of this thesis would not have been possible without the support of weekly-diss: Kevin Rich, Bob Estes, and Phil Nico

My dissertation committee members, Carole McNamee and Matt Farrens, provided much feedback that has made this thesis easier to read.

My brother, Dan, has always been there. Thanks dude. My parents have provided continued support for my education and all my endeavors. Thanks Mom and Dad. Many friends have helped make the last ten years fun, relaxing, and sometimes a little dangerous. Please accept this blanket thanks.

I love you Laura. Thank you for everything.

Chapter 1

Introduction

Concurrent programming has existed since the early days of computing, first to manage asynchronous devices, later to exploit parallel processors. Fundamental to a concurrent program is the notion of a *thread* or *process*, which is an independent execution stream. Whereas a sequential program consists of a single thread, a concurrent program consists of two or more threads that work together to perform a task.

Even sequential programs have some concurrency happening behind the scenes. This concurrency is enabled by the operating system and is not directly visible to a sequential program. For example, an interrupt handler is often used to retrieve input from the keyboard or a timer interrupt handler updates the system clock. It is not surprising that operating systems are highly concurrent programs that often have a variety of mechanisms for synchronizing both user programs and different system services.

Until recently, most concurrent programs were either operating systems or parallel programs for scientific or numerical computing. More and more programmers are taking advantage of concurrency for managing graphical user interfaces, input/output, and client/server programs. In addition, concurrency is becoming more accessible through increased operating system support for threads, the availability and standardization of libraries (e.g., POSIX Threads [64] and the Win32 interface for threads [28]), language mechanisms for threads (e.g., Java threads [52] or SR processes [10]), and the increasing availability of low-cost multiprocessors (e.g., SMP x86 machines and SMP Sparc servers).

Despite the increase in support for concurrency in libraries and programming languages, concurrent programming remains difficult. Programmers are still responsible for preventing race conditions and deadlock, while at the same time ensuring good performance both on uniprocessors and multiprocessors.

To write a concurrent program, one needs access to mechanisms for thread creation, synchronization, and communication. Thread communication is achieved via shared variables or message passing. Concurrency mechanisms are offered both in the form of library calls, which might be implemented in terms of operating system primitives, and as part of some programming languages. Thread libraries, such as Pthreads or Win 32/NT threads, augment sequential languages, such and C and C++, with the necessary functions to program concurrency. Using a thread library allows a programmer to leverage off his or her existing knowledge of a sequential language for writing concurrent programs. However, such libraries are fairly primitive and lack good integration with the host language. In this sense, thread libraries provide rather low-level mechanisms for programming concurrency.

Many high-level languages — such as SR, Java, Modula-3 [86] and Ada [20] — have direct semantic support for concurrency. Concurrency is a fundamental part of these languages; therefore, these languages not only provide high-level concurrency mechanisms, but all the other language elements are designed with concurrency in mind. Often, this type of integration of language and concurrency simplifies the task of writing parallel and distributed programs by providing a clean concurrency model and simplifying thread synchronization and communication. A clean syntax and semantics for concurrency allows the programmer to concentrate on the basic problems of mutual exclusion, deadlock, and efficiency. In addition, incorporating concurrency in the semantics of a language reduces the complexity of developing a formal logic for the language (which can be used to construct correctness proofs of concurrent programs). In the library approach, not only must a programmer use (sometimes complex) function calls, but he or she must also contend with the changes in the semantics of a sequential language introduced by concurrent execution (e.g., simultaneous access to global variables and function call reentrancy).

Programming languages with semantic support for concurrency can help manage the complexity of programming modern platforms[1]. However, as parallel and distributed platforms become more diverse, it becomes increasingly difficult to develop efficient and portable compilers and run-time systems. Concurrent languages rely on a run-time system to provide support for threads and, for some languages, communication.

[1]The term *platform* is used to denote a hardware and software combination that makes a complete computer system, including the processor architecture, bus architecture, memory system, operating system, supporting libraries, and possibly the network interconnect. In the case of a bare machine, a platform refers to just the hardware configuration and possibly some minimal support software.

A language run-time system serves as the glue between the compiler-generated code and the underlying platform. In addition to having a significant impact on performance, run-time systems are also largely responsible for the degree of portability of a language implementation. However, the increasing diversity in parallel and distributed platforms, both in hardware architectures and in operating systems, is creating a growing gap between language implementations and target platforms, making it increasingly difficult to develop efficient and portable compilers and run-time systems. Concurrent programming languages are particularly dependent on threads and communication, both of which can vary greatly from platform to platform.

This dissertation examines the growing gap between concurrent languages and the underlying platform. The question is: How should language-specific thread mechanisms be implemented in compilers and run-time systems to best support concurrent programming languages? Current approaches are either ad-hoc and specific to a particular language or run-time system, or use traditional software layering to handle the complexity of managing multiple targets but sacrifice performance.

We have gained considerable experience implementing a particular concurrent programming language, SR, on several different platforms. Based on this experience, we have evaluated several different techniques for mapping language-level concurrency to the underlying operating system and architecture. The implementation experience and evaluation has resulted in the *Mezcla thread framework*, a new approach to incorporating threads into concurrent run-time systems.

Unlike much of the previous research in improving the performance of concurrency mechanisms, which has been limited to high-performance parallel computing, this work focuses on general purpose programming languages for both applications and system software. Now that an increasing number of programmers are taking advantage of concurrency, it is clear that we need to improve the state of the art in implementing concurrent programming languages designed for systems programming and general-purpose applications, as opposed to just scientific and numeric applications. However, the techniques developed in this dissertation should carry over to scientific and numeric applications and their corresponding programming languages.

The remainder of this chapter introduces fundamental concepts, assumptions, and nomenclature on which the rest of this dissertation is based. In addition, this chapter summarizes the main contributions of and gives a road map to this dissertation.

1.1 Parallel and Distributed Hardware

Advances in concurrent programming have been partly driven by the development of parallel and distributed hardware. The distinction between "parallel" and "distributed" is becoming somewhat blurred. However, in general, a parallel machine is one that has processing elements that are tightly coupled by a bus or interconnect. The processors in a parallel machine may or may not share the same memory. A distributed machine is one that has processing elements that are loosely coupled by some type of network. The processors in a distributed machine typically do not share memory and have a higher and less predictable transfer latency. Other hardware designs that are considered parallel processors, but do not fall into the above categories, include vector machines and data-flow machines.

While researchers have produced a large number of experimental parallel machines, this dissertation focuses on commodity hardware. Many research efforts have focused on building languages, compilers, and run-time systems for experimental and specialized parallel machines. Fewer research efforts have focused on general purpose concurrent languages on commodity hardware. Both areas are significant, but now that more and more programmers are taking advantage of concurrency, it is important to improve the state of the art in compilers and run-time systems for concurrent languages on stock hardware. In addition, the performance of commodity hardware has been increasing at a much faster rate than specialized machines because of greater demand, which compounds the need to develop new implementation techniques.

We view commodity hardware as a collection of uniprocessor or multiprocessor workstations joined by a high-speed interconnect. Shared-memory multiprocessors are now cost effective and are keeping pace with the fastest processors on the market (e.g., several companies have commercial multiprocessors that use some of the latest chips, such as Intel's Pentium II, Compaq's Alpha, SGI's MIPS R10000, and Sun's UltraSparc). Therefore, it is not necessary to sacrifice sequential performance in order to take advantage of a multiprocessor. Similarly, high-speed network technology is becoming accessible to mainstream markets. Network switches and interconnects such as asynchronous transfer mode (ATM), Myrinet, and Gigabit ethernet all push network performance well beyond standard 10Mbit ethernet at affordable prices.

Of course, most commodity hardware started as experimental before moving into the mainstream. It is difficult to predict what the next generation of commodity machines will look like since hardware technology is a moving target and often business decisions and marketing play a large role in defining "commodity hardware". However, some emerging technologies appear to have promise.

One way to take advantage of increasing chip density is to put more than one processor on a single chip [87]. Such a configuration has the potential to greatly reduce the processor to processor communication cost and can simplify shared cache design. However, programming a multiprocessor on a single chip poses all the problems found on shared memory multiprocessors, such as synchronization and load distribution.

Another emerging trend in processor design is the multi-threaded architecture [36]. A multi-threaded processor provides hardware support for multiple execution streams (traditionally, processes and threads are completely managed by the operating system). The advantage of such a design is that the processor can quickly schedule threads to hide memory latency. The challenge of such architectures is developing compilers that can decompose arbitrary programs into multiple threads so that the processor always has something to run.

Finally, distributed shared memory (DSM) [77] has proven to be a useful programming model, but general-purpose software-based implementations have significant performance problems. Hardware-based implementations of a shared physical address space may prove to make DSM a reality for general-purpose computing [36]. However, even hardware-based approaches will require programming languages and their implementations to properly exploit underlying hardware mechanisms.

1.2 Concurrent Programming

Concurrent programming spans a wide range of techniques, mechanisms, and languages. The central problem in concurrent programming is the coordination of multiple streams of execution. We will refer to a single stream of execution as a *thread* or a *process*. A concurrent program usually consists of two or more threads, which may or may not reside on the same computer. Threads that execute on the same computer can usually share the same memory, and thus can use shared memory or shared variables to communicate and synchronize. A concurrent program may also consist of threads that reside on different computers, in which case, threads must communicate and synchronize by sending and receiving messages over a network.

Compared to sequential programming, concurrent programming can be considerably more complicated. Shared data must be accessed in a coordinated manner to ensure program correctness. Synchronization mechanisms enable concurrent programs to control access to shared data, but they also introduce the possibility of deadlock, starvation, and inefficiency. The programmer's goal is to write concurrent programs that are both correct and efficient.

Research in concurrent programming has generated a large number of mechanisms and languages to simplify the task of writing concurrent software. Such developments have made it easier to write concurrent software. Chapter 2 takes a closer look at some fundamental concurrency mechanisms and languages.

1.3 Language Implementation

The implementation of a concurrent programming language usually consists of a *run-time system* (RTS)[2] in addition to a compiler. Run-time systems are used to simplify code generation, to implement the dynamic aspects of complex language elements, and to interface with the underlying operating system and/or architecture. Often, the gap between the semantics of the language and the underlying platform is too large to accommodate direct translation from source code to object code. As such, a run-time system provides interfaces for manipulating the internal representation of language elements. The compiler's code generator is then responsible for producing code that properly invokes the run-time system interfaces. Indeed, many sequential languages require run-time support for all of these same reasons, but concurrent languages typically require additional extensive support for threads and communication.

Run-time systems have a large impact on both the performance and portability of programs written in concurrent programming languages. While performance is often a primary concern for language implementations, portability is becoming increasingly more important as the target platforms for parallel and distributed computing grow more diverse. Although current trends in the hardware market indicate convergence towards networks of workstations and shared-memory multiprocessors, the range of software support (i.e., thread packages, communication interfaces, and operating system kernels) remains in flux. In addition, portability is often a reason to use a concurrent programming language (e.g., Java is used to write multi-threaded applets that can run in almost any browser on any

[2]We use RTS to denote both "run-time system" and "run-time support."

desktop computer), but program portability is limited to the actual number of platforms supported by the language implementation.

Despite the importance of run-time systems, there are very few established techniques for their implementations. Most language implementations use ad hoc approaches for constructing run-time systems and they cannot be reused from one language to another. The current, most popular technique uses traditional software layering to hide architectural and operating system dependencies. However, layering often leads to performance problems. Chapter 4 analyzes the implementation of the SR run-time system using several different techniques and Chapter 5 evaluates existing techniques for incorporating threads into concurrent run-time systems.

1.4 The Mezcla Approach

This dissertation introduces Mezcla, a new approach to implementing concurrent run-time systems. At a very high level, Mezcla is a design philosophy. The basic idea is to use code generation techniques to create custom thread operations based on the requirements of a particular language and its run-time system. Mezcla is an attempt to move away from treating a run-time system as a fixed library or even a parameterized library. Instead, we believe that run-time systems should be specialized based on language requirements.

We explore the Mezcla design philosophy by applying it to the development of thread components for concurrent run-time systems. We develop the *Mezcla thread framework* for generating specialized thread systems that depend on the requirements of a programming language and the functionality of a target platform. We show that the Mezcla design approach can achieve both high-performance and retargetability when implementing threads in concurrent run-time systems.

1.5 Contributions

This dissertation presents three contributions to the field of language implementation: lessons learned from extensive implementation experience with the SR concurrent programming language, an evaluation of current thread implementation techniques, and the Mezcla thread framework, a new approach for incorporating threads into concurrent run-time systems.

1.5.1 Implementation Experience

The first contribution of this dissertation is to explore the implementation issues of a particular concurrent programming language, SR, on a diverse range of target environments, including microkernels, parallel run-time libraries, POSIX threads, and a bare hardware implementation. Our experience reveals many important factors in run-time system design, and at the same time evaluates several target platforms from both performance and software engineering perspectives.

1.5.2 Thread Evaluation

The second contribution of this dissertation is to classify and analyze current techniques for implementing threads in concurrent run-time systems. This analysis will help future language implementors understand the advantages and limitations of current approaches.

1.5.3 The Mezcla Thread Framework

The third and most significant contribution of this dissertation is the *Mezcla thread framework*, which addresses the tension between portability and efficiency found in current thread systems. Many thread packages support only a limited number target configurations and are generally inflexible with respect to scheduling. Even configurable thread packages distance core thread operations and thread state from client code. In addition, most thread implementations duplicate some of the functionality found in concurrent language run-time systems. The Mezcla thread framework addresses these limitations by enabling a language implementor to generate specialized, light-weight thread systems based on the requirements of the run-time system and the functionality of the target platform.

1.6 Organization

The rest of this dissertation is organized as follows. Chapter 2 provides background material on concurrent programming, thread systems, and concurrent programming languages; it serves as a foundation for the latter chapters. Chapter 3 presents the SR concurrent programming language and the original implementation. Chapter 4 describes early implementation experience with SR on a wide variety of platforms using different implementation techniques. The work presented in this chapter provides supporting evidence for the Mezcla approach. Chapter 5 evaluates current techniques for incorporating threads into run-time systems. Chapter 6 presents several design issues for the Mezcla thread framework. Chapter 6 walks through an example of using the prototype implementation of the Mezcla thread framework and Chapter 8 gives a detailed description of the prototype. Chapter 9 provides a performance evaluation of the Mezcla prototype. Finally, Chapter 10 offers conclusions and directions for future research.

Chapter 2

Background

This chapter provides background material in four areas. First, it introduces concurrent programming, including some of the most common and classic concurrency mechanisms. Second, it outlines the advantages of concurrent programming languages and presents the concurrency requirements of several programming languages and the implications these requirements have on implementations; some of these implications are revisited in Chapter 5 in the context of specific run-time systems. Third, it discusses the basic implementation issues for concurrent programming languages. Finally, it describes some of the most prevalent libraries and operating system interfaces for threads.

2.1 Concurrent Programming

As described in Section 1.2, concurrent programming comprises a large number of techniques, mechanisms, and languages. We view concurrent programming as the problem of coordinating multiple *threads* of control or *processes* to perform some task. These threads may or may not reside on the same computer. Threads that execute on the same computer can usually share the same memory, and thus can use shared memory or shared variables to communicate and synchronize. A concurrent program may also consist of threads that reside on different computers. In this case, threads must communicate and synchronize by sending and receiving messages over a network. Our view of concurrent hardware is illustrated in Figure 2.1.

A multiprocessor supports true parallelism, while uniprocessors support pseudo parallelism through multiprogramming. In either case, the model is the same: several

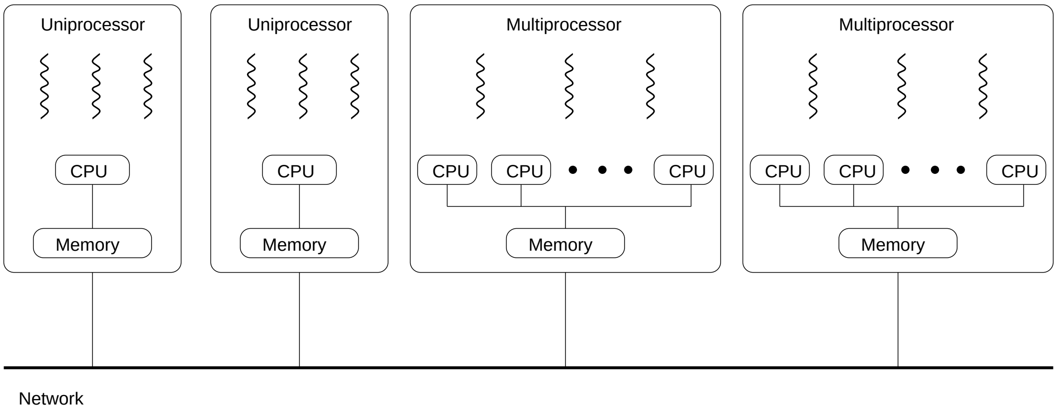

Figure 2.1: Concurrent Hardware (squiggles represent threads)

sequential threads running in parallel. To do useful, concurrent work, these threads must have some method of exchanging information. Many years of research have produced a wide variety of mechanisms for enabling multiple threads to have coordinated access to shared data. These mechanisms are summarized below; for complete coverage of this material, including examples of languages that provide these mechanisms, see [8].

2.1.1 Busy Waiting

An extremely low-level technique for synchronizing access to shared data is called *busy waiting*. As its name implies, busy waiting consists of a loop that repeatedly checks a shared variable for a particular value. As a simple example, the variable might be a boolean value that indicates whether or not another thread is in a critical section. For proper synchronization, it is necessary to have atomic access to the synchronization variable. There are both hardware [6] and software [38, 89, 93] solutions that guarantee atomic access to shared data.

2.1.2 Semaphores

On systems that support multiprogramming, busy waiting is not usually an acceptable method for synchronizing threads because it wastes CPU time. Only on true multiprocessors running dedicated threads is it reasonable to use busy waiting. A slightly higher-level mechanism for coordinated access to shared data is the *semaphore* [40]. The semaphore is an abstract data type that is accessed through two operations: **P** and **V**. The semaphore data type consists of an integer variable, s, whose value is always greater than or equal to zero, and a queue of waiting threads. If s is greater than 0, the **P** operation decrements s by one and allows the thread invoking **P** to continue execution; otherwise, the thread must block on the semaphore queue. The **V** operation performs one of two actions. If **V** is invoked and one or more threads are blocked on the semaphore, then one of the threads — chosen deterministically, usually first-come first-served — will gain possession of the semaphore and that thread will unblock. If no threads are blocked on the semaphore, then **V** simply increments s. Semaphores in which s can increase without bound ($0 \leq s$) are called *general semaphores*. If s is bounded by one (i.e., $0 \leq s \leq 1$), then s is called a *binary semaphore*. When using a binary semaphore, it is possible for **V** to block.

2.1.3 CCRs and Monitors

Higher-level mechanisms for synchronizing access to shared data include the *conditional critical region* (CCR) [58] and the *monitor* [59]. The CCR denotes protected data that can only be accessed within designated regions that guarantee mutual exclusion. Access to the protected data is synchronized via boolean conditions that are checked at the beginning of the regions. The monitor, on the other hand, is a module that encapsulates an abstract resource. A monitor consists of private data, which can only be accessed by the operations defined in the monitor, and guarantees that no two threads can be executing code in the monitor at the same time. Thread synchronization is handled by the **signal** and **wait** primitives, which operate on condition variables (abstractions of waiting queues).

2.1.4 Message Passing

The mechanisms mentioned above — busy waiting, semaphores, CCRs, and monitors — assume that threads have access to shared data. Where several computers are connected by a network with threads executing on different machines, sharing data may not be possible or efficient. The most common form of distributed communication is *message passing*. In its basic form, message passing consists of two primitives: **send** and **receive**. The arguments to **send** include a destination (e.g., a port or a mailbox) and the actual data to be sent. Likewise, **receive** designates a source from which a message is to be acquired and a variable (or variables) to which the data contained in the message will be assigned. Message passing can have many different semantics; two common forms are *asynchronous message passing* and *synchronous message passing*. In asynchronous message passing, a thread invoking **send** will immediately continue execution, while in synchronous message passing, **send** will block until the message has been received (and in some cases acknowledged) by another thread. It is common in both asynchronous and synchronous message passing for **receive** to block until a message exists.

2.1.5 Remote Procedure Call and Rendezvous

Message passing is a relatively low-level mechanism for communication between threads. More natural mechanisms are the *remote procedure call* (RPC) and *rendezvous*.

RPC allows a thread to make a call to a procedure serviced by another thread, which is not necessarily located on the same machine. Conventional RPC requires a new thread to be created to service the procedure call. Because the procedure call is an essential component of most languages, RPC is well understood and easy to use. In its most common form, the calling thread will block until the remote thread is done executing the named procedure. However, RPC must deal with potential communication failures; thus, special semantics are required to deal with such failures [30].

Rendezvous is similar to RPC in that it is invoked with a procedure call, but unlike RPC, the remote thread is already running either waiting to accept invocations, currently servicing another rendezvous request, or performing some unrelated computation. A new thread is not created.

2.1.6 Distributed Shared Memory

Another technique for exploiting parallelism in a distributed system is to provide a group of threads a single, shared address space. This technique, usually called *distributed shared memory* (DSM) [77], is attractive because it allows the programmer to use a more conventional model of computation. Because communication is implicit, only synchronization primitives such as semaphores and monitors are required to coordinate access to shared variables. However, efficiently implementing DSM has proven to be difficult. First, DSM requires a run-time system that guarantees *memory consistency*. When a thread changes a portion of DSM, it is important that all other threads accessing the same DSM see the change; otherwise, the program will most likely produce incorrect results. Maintaining memory consistency ensures that all threads see the same changes to shared memory. Second, the burden of data placement is now placed on the compiler rather than the programmer. While the compiler technology for languages that support DSM is improving, it is still difficult for a compiler to handle the general case.

DSM is one of several related techniques based on distributed shared data. For instance, Linda [5] provides the concept of the *tuple space*. The tuple space is essentially an atomically-accessed shared database. The Orca language [16] features distributed shared data structures. Several other languages provide concurrency through shared objects, including ALPS [114], ConcurrentSmalltalk [119], and Emerald [31].

2.2 Concurrent Programming Languages

To simplify the process of writing concurrent programs (both parallel and distributed), researchers have designed many different concurrent programming languages [17]. A concurrent language is based on a model of computation, which is manifested in a language's primitives for specifying parallelism, communication, and synchronization. Different concurrent programming models include, but are not limited to, message passing, shared memory, and distributed objects.

This section discusses the benefits of using concurrent programming languages, then examines specific languages and language extensions. Although many concurrent languages exist, the languages described here were chosen for three reasons. First, the languages are representative of their respective classes. Second, most of the languages have introduced novel and useful constructs. Finally, there exists a substantial number of publications that document the languages described below. Note that some of these languages and many more are reviewed in [17].

For each language, we describe the main synchronization and communication mechanisms. In addition, for each language, we provide an overview of one or more implementations and discuss some key implementation issues. We focus on the following thread mechanisms: creation, synchronization, scheduling, and communication (remote). Because of its central role in this dissertation, we defer a more comprehensive description of the SR programming language and its implementation until Chapter 3.

2.2.1 Benefits

No one model for concurrent programming is universally accepted because different models are best suited for different applications. Regardless of the model used, studies show that it is easier to write concurrent programs in a programming language that supports a concurrent programming model, than to use a sequential language augmented with library calls for communication and synchronization [32, 99]. Simplification of the coding process is only one of the many benefits of developing programs in a concurrent programming language; a complete list of desirable properties are listed below.

- **Expressiveness** It is easier to write concurrent applications in a concurrent programming language. By providing primitives for concurrency, communication, and synchronization, development time is spent on program functionality and not on the details of low-level libraries or kernel interfaces.

- **Conversation Management** Communication is central to distributed programs, therefore language mechanisms that facilitate interprocess communication (conversations between threads) simplify the programming effort.

- **Interface Definition** It is easier to define server interfaces. Many concurrent programming languages are inherently modular (i.e. SR [11], Orca [16], and Ada [112]). Such languages provide a clean syntax and semantics for writing interface specifications.

- **Type Checking** It is possible for a compiler to catch type errors in the use of concurrency primitives because a concurrent programming language incorporates communication and synchronization primitives into the syntax and semantics.

- **Portability** It is possible to write more portable programs using a concurrent programming language than using a language that uses library calls. Generally, a concurrent language presents a higher level of abstraction, which facilitates portability.

- **Error and Exception Handling** It is more robust to implement error and exception handling in the run-time support than to write code that handles all possible error conditions

- **Optimization** It is possible to optimize high-level concurrency mechanisms because objects such as communication channels have first-class status.

- **Verification** It is easier to reason about programs written in a concurrent programming language because the communication and synchronization primitives are defined in the language's semantics.

Some languages can be classified as *implicitly parallel* in which a programmer writes in a sequential style, but a compiler extracts parallelism from a program. In contrast, an *explicitly parallel* language requires the programmer to directly manage multiple threads of control. Implicitly parallel languages shift the complexity of managing concurrency from the programmer to the compiler, but mainly target numerical or symbolic applications. For example, one would not write an operating system in an implicitly parallel language, at least not yet. Other programming models provide alternative means for expressing or obtaining parallelism. Two other models include parallel functional [60, 61, 108] and concurrent logic [100]. Languages based on these models have interesting features, but do not fall within the scope of this research. This dissertation focuses on languages that support explicit concurrency.

Languages also exist that support fault tolerance (either implicitly or explicitly) and atomic transactions. NIL [105], for example, was designed for creating reliable distributed programs. Fault tolerance in NIL is handled implicitly by the run-time system. Another example is Argus [80], a language based on atomic transactions. The two main features of Argus are guardians and actions. Guardians are program modules that can withstand crashes, while actions are sets of atomic executions.

2.2.2 Java

Java is an object-oriented language loosely based on C++. Most of the expression syntax of Java is similar to that of C++, but Java lacks some of the more complicated features of C++, such as multiple inheritance. Java does not have arbitrary memory pointers; instead, it has object references. Unused memory is garbage collected.

The fundamental structure in the Java language is the *class*. A class encapsulates both data and the methods that operate on the data. New classes can be declared using inheritance. All Java programs consist of one or more classes. Classes instances are created dynamically during program execution.

Java programs can have multiple threads. New threads are created by instantiating classes that either inherit the `Thread` class or implement the thread class interface. Thread execution does not begin until the `start` method is invoked on the new object.

Synchronization is achieved by declaring member functions of the class or code blocks as `synchronized`. Objects that are instantiated from classes with synchronized functions or blocks are called *synchronized objects*. Threads that invoke methods in a synchronized object are guaranteed to have exclusive access to the object. If more than one thread attempts to access a synchronized object, one of them will be blocked until the other thread leaves the object. Synchronized objects are similar to monitors (see Section 2.1.3). They support conditional synchronization with the `notify` and `wait` keywords. Figure 2.2 shows the code for a Java class that implements semaphores.

Thread scheduling in Java is highly unspecified and is left to the implementation. For example, whether scheduling is preemptive or non-preemptive is not specified by the language. It is also not possible for a Java program to modify or determine the scheduling policy on a particular implementation. Java does have the notion of thread priorities, but these are only treated as "hints".

Remote communication in Java is very primitive. A class library provides access to TCP or UDP sockets. A slightly higher level of communication is achieved using a type of RPC (see Section 2.1.5) called remote method invocation (RMI). To use RMI, a special compiler is needed to generate client and server code to handle remote invocations.

A Java compiler translates Java source code into bytecode. A Java Virtual Machine (JVM) is required to execute Java bytecode. A JVM will either interpret the bytecodes directly or compile the bytecode dynamically to the target architecture. In either case, the JVM is really a run-time system for Java programs. The JVM provides support for threads, garbage collection, and I/O. It is the JVM that ensures the portability of Java programs.

2.2.3 Modula-3

Modula-3 [86] is a descendant of Modula-2, with many enhancements. Like Modula-2, Modula-3 was designed for systems programming. Modula-3 maintains the concept of a module, but also supports module inheritance and generics. The language is also type safe and uses garbage collection to reclaim unused objects. In addition, Modula-3 supports threads and synchronization through modules. Similar to Pthreads, Modula-3 has mutex and condition variables. However, Modula-3 also supports exception handling and alerts to interrupt long running computations. Figure 2.3 shows a Modula-3 object for implementing

```
public class Semaphore {

  public Semaphore(int initValue) {
        value   = initValue;
        blocked = 0;
  }

  public synchronized void P() {
        if (value > 0) {
            value--;
        }
        else {
            // wait
            blocked++;
            while (value == 0) {
                try { wait(); }
                catch(InterruptedException e) {}
            }
        }
  }

  public synchronized void V() {
        if (blocked == 0) {
            value++;
        }
        else {
            // signal
            blocked--;
            notify();
        }
  }

  private int value = 0;
  private int blocked = 0;
}
```

Figure 2.2: A Java Semaphore Class

semaphores.

Similar to Java, Module-3 supports type safety. This feature has been used by the SPIN operating system [27] to support application-specific kernel extensions.

Remote communication in Modula-3 is supported by libraries. In particular, a set of tools and libraries is available to support *distributed objects*. Distributed objects are similar to Java's RMI (remote method invocation) facility.

The Modula-3 run-time system is responsible for dynamic type checking, garbage collection, and thread support. Many current implementations of Modula-3 are built on top of Pthreads. The language specification does not dictate how threads are to be scheduled; this decision is left to the run-time system implementation.

2.2.4 Ada 95

Ada 95 [65] is a general-purpose programming language designed for writing large, reliable applications. Ada 95 is a descendent of Ada [112] and supports object-oriented programming, enhanced tasking facilities, and better support interoperability between different languages and systems. In addition, Ada 95 defines several optional "annexes" for supporting different application areas. These annexes include systems programming, real-time systems, distributed system, informations systems, numerics, and safety and security.

Ada 95 contains fundamental types and control structures that can be found in most Algol-like languages. Programming in the large is supported with packages, which encapsulate data and procedures. The programmer can control which elements of the package are visible to the rest of a program. The package construct can be extended to support objects and classes. That is, new classes can be formed by inheriting and extending existing classes. Packages can also be constructed using generics.

Concurrency in Ada 95 is achieved with the *task* construct. A task definition is similar to a package definition, except that it specifies a new thread of control. Using this construct an Ada 95 program can be made up of several interacting tasks. The main form of communication between tasks is through *rendezvous* (see Section 2.1.5). A server awaits communication with the **accept** statement and a client initiates communication by invoking a task's entry point. Multiple entry points to a server can be coordinated with the **select** statement.

Unlike original Ada, Ada 95 also provides *protected objects*, which combine func-

```
MODULE Semaphore;
(* Semaphore object implementation *)

  IMPORT Thread;

  REVEAL
    T = Public BRANDED OBJECT     (*make the semaphore type unique*)
          s: CARDINAL;            (*initial value of semaphore*)
          w: Thread.Condition;    (*queue*)
        OVERRIDES
          init:= Init;
          P:= Test;
          V:= Leave;
        END; (*T*)

  PROCEDURE Init(sem: T; initialValue: CARDINAL := 1): T =
  BEGIN
    sem.s:= initialValue;
    sem.w:= NEW(Thread.Condition);
    RETURN sem;
  END Init;

  PROCEDURE Test(sem: T) =
  BEGIN
    LOCK sem DO
      WHILE sem.s = 0 DO Thread.Wait(sem, sem.w) END;
      DEC(sem.s);
    END; (*LOCK*)
  END Test;

  PROCEDURE Leave(sem: T) =
  BEGIN
    LOCK sem DO
      INC(sem.s);
      Thread.Signal(sem.w);
    END; (*LOCK*)
  END Leave;

BEGIN
END Semaphore.
```

Figure 2.3: A Module-3 Semaphore Object

tionality found in monitors and CCRs (see Section 2.1.3). Figure 2.4 shows a protected object that implements a semaphore (this example is taken from [20]). The *entry* serves as a barrier. If one or more tasks are blocked on an entry, the condition specified by the entry is checked each time a task leaves the protected object, thus allowing blocked tasks to continue.

```
protected type Semaphore(Start_Count: Integer := 1) is
  entry Secure;
  procedure Release;
private
  Count: Integer := Start_Count;
end Semaphore;

protected body Semaphore is

  entry Secure when Count > 0 is
  begin
    Count := Count -1
  end

  procedure Release is
  begin
    Count := Count + 1
  end Release

end Semaphore
```

Figure 2.4: An Ada 95 Semaphore as a Protected Object

In addition to the basic form of tasks and protected objects, Ada 95 also supports timed and condition calls, exceptions, and abort semantics on server invocations. Scheduling policies are determined by a mix of programmer control (through *pragmas*) and the underlying implementation. Ada 95 supports different types of scheduling support through different implementations. The real-time systems annex enhances the baseline scheduling policy with support for priorities, locking policies, and queuing policies.

True distributed communication is achieved with the distributed systems annex. This annex allows a program to be split into different partitions, which can be active or passive. A passive partition has no thread of control and simply provides sequential library code. An active partition has its own copy of the run-time system and can communicate with other active partitions. Package interfaces can be annotated to allow remote procedure calls between partitions via the remote call interfaces.

The Ada 95 run-time system is responsible for garbage collection, tasking (i.e., creating and scheduling tasks), and remote communication. However, unlike some of the other languages described in this section, Ada does not require that its run-time systems support tasking and remote communication. The idea is that a sequential program that does not use tasking should not have to pay for the extra overhead to implement it. Of course, a program that utilizes tasking will require a run-time system with such support. Some current implementations of Ada (e.g., GNAT [95]) use Pthreads to implement tasking.

2.2.5 Lynx

The Lynx programming language [97, 98, 99] was designed for writing distributed applications and system programs. Unlike languages that support communication only between processes in the same program, Lynx allows type-safe message passing between loosely coupled distributed programs. Lynx supports dynamic communication pathways through a *link* abstraction. A link is a dynamic, first-class object that allows bidirectional communication between processes. Links can be created, destroyed, and passed in messages. Due to the first-class nature of a link, arbitrary communication configurations can be created at run-time. The Lynx syntax, basic data types, and scope rules are similar to those of Modula-2 [117].

Figure 2.5 shows a simple solution, taken from [98], to the producer/consumer problem written in Lynx. The producer and consumer processes are two separate programs. The producer and consumer programs must be joined by a link that is created by a parent program (see Figure 2.6). A parent program will create a new link, then pass one end of the link to the producer and the other end to the consumer. The parent program will also create a new process for each program, which may or may not be located on the same machine. The **entry** statement is used to define a template for a remote operation. In the producer process, the main begin-end body uses the **connect** statement to request a remote operation on the consumer link. In the consumer process, the **accept** statement is used to service an operation request on the producer link. In this example, **connect** and **accept** allow the two processes to rendezvous.

```
process producer (consumer : link);

type data = whatever;
entry transfer (info : data); remote;

function produce : data;
begin
    -- whatever
end produce;

begin -- producer
    loop
        connect transfer (produce |) on consumer;
    end
end producer
```

```
process consumer (producer : link);

type data = whatever;
entry transfer (info : data); remote;

function consume : data;
begin
    -- whatever
end consume;

var buf : data;

begin -- consumer
    loop
        accept transfer (buf) on producer;
        reply;
        consume (buf)
    end
end consumer
```

Figure 2.5: The Producer/Consumer Problem in Lynx

The programs shown in Figure 2.5 are relatively simple, but they do give a feel for the language. Some of the features not illustrated in this example include implicit receipt (similar to RPC), thread creation, link movement, and exception handling. A Lynx program can have multiple threads of execution, possibly created explicitly with the **call** statement, or by implicit receipt on a link. However, threads are not executed concurrently, so they are more like coroutines. The Lynx run-time package handles the transfer of control when a thread blocks. Since only one thread executes at any time, explicit synchronization to shared data is not needed.

```
var L:link;
begin
    startprocess("consumer", newlink(L));
    startprocess("producer", L);
    ...
```

Figure 2.6: The Producer/Consumer Parent

Lynx has been implemented on the University of Wisconsin's Crystal multicomputer running the Charlotte distributed operating system [42] and on the BBN Butterfly Parallel Processor running the Chrysalis operating system [98]. In addition, a paper design for Lynx exists for the SODA distributed operating [69]. The most significant part of the Lynx implementation is the run-time system, which manages the link abstraction, included dynamic creation, destruction, and run-time type checking. Furthermore, the run-time system schedules threads and coordinates exception handling.

In [98, 99], Scott presented some interesting observations based on his experience with different Lynx implementations. Originally, Lynx was designed for the Charlotte operating system, which provides a link abstraction similar to the link abstraction in the language. In fact, the motivation for Lynx was based on the Charlotte link abstraction. Unexpectedly, the Charlotte implementation of Lynx turned out to be quite difficult. The Charlotte primitives seemed ideal, but presented complications because of their high-level nature. The lower-level primitives provided by SODA and Chrysalis simplified their implementations and realized better performance. As a consequence of this observation, Scott recommends that most language functionality should be implemented in a run-time system that exists outside of the operating system kernel.

2.2.6 Orca

Orca [15, 18, 16] is a distributed programming language based on the shared data-objects model and is designed for application programming, not for system-level programming. Rather than providing complete distributed shared memory, Orca extends the abstract data type model by allowing encapsulated resources to be shared by one or more processes. Processes can only access the shared data type through specified operations, thus synchronizing access to the shared data-object. Orca has no global data-objects, so to facilitate sharing, a process creates new processes with data-objects as parameters.

Figure 2.7 shows an Orca program from [19] that implements a distributed queue of integers. All Orca objects have a *specification* part and an *implementation* part. The IntQueue object has two operations: `append` and `remove_head`. In addition to the basic types, Orca has built-in data types for lists and graphs (they are needed to represent recursive structures, as Orca does not have pointers). Any operation on a single data object is executed indivisibly; thus, the `append` operation does not need any additional

```
object specification IntQueue;
    operation append (X: integer);
        # append X to the queue
    operation remove_head(): integer;
        # wait until queue is not empty;
        # remove and return head element
end;
```

```
object implementation IntQueue;
    Q: list of integer; # internal representation

    operation append(X: integer);
    begin # nonblocking operation
        add X to end of Q;
    end;

    operation remove_head(): integer;
        R: integer;
    begin
        guard Q not empty do
            R := first element of Q;
            remove R from Q;
            return R;
        od;
    end;

begin
    Q := empty; # initialize IntQueue object
end;
```

Figure 2.7: Queue of Integers in Orca

synchronization to guarantee the consistency of `Q`. The `remove_head` operation uses the **guard** statement to ensure `Q` is not empty, therefore, **guard** is a synchronization primitive. The **guard** statement will cause the process that invoked `remove_head` to block if `Q` is empty. The process will remain blocked until another process adds an element to `Q` using `append`.

Because Orca is based on the shared data-object model, communication between processes is implicit, and therefore no communication primitives are needed. The burden of distributing data-objects is placed on the run-time system, so the efficiency of the run-time system is critical to the performance of any Orca program.

The Orca run-time system uses a data consistency protocol to ensure that all shared data-objects are always in a consistent state. The Orca implementors experimented with various protocols and found the most successful protocol is one based on reliable broadcast [67]. It replicates each data-object on each processor and then broadcasts the new object state when it is modified. By using a reliable broadcast mechanism, the run-time system can easily serialize all operations on shared data-objects, which guarantees that each process always has the same "view" of a shared data-object. Because the Orca run-time system need not broadcast on read operations, the reliable broadcast method performs best on applications that read objects frequently, but modify them infrequently.

While Orca programs can run on shared-memory multiprocessors, the language and its implementation are both oriented toward distributed hardware. Shared data-objects require considerable overhead to be used to synchronize threads running on the same machine. In addition, the Orca run-time system uses a single "big lock" to synchronize access to internal data. This coarse-grain solution to run-time system synchronization reduces the opportunity to take advantage of true parallelism on shared-memory multiprocessors.

The Orca run-time system has several implementations: one on a 68000-based multiprocessor, one on the bare hardware of a 68000-based network of computers, one on top of the Amoeba distributed operating system [109], and one on the Panda portability layer [29]. The Panda implementation is the most recent and is an attempt to achieve greater portability across several platforms. Panda provides interfaces for threads, message passing, remote procedure call, and reliable group communication (which is essential to the Orca run-time system).

2.2.7 Hermes

Hermes is a high-level distributed programming language developed at IBM T. J. Watson Research Center [104] that evolved from the NIL (Network Implementation Language) programming language [105]. Designed for building large distributed systems, Hermes supports dynamic, long-lived, distributed components. Hermes is based on a *process* model. Processes are independent entities, which do not share data with any other process. Processes are also the active entities in the language and replace the notion of a procedure. Furthermore, processes interact through typed interprocess communication using a *port* abstraction. Ports are categorized as *inports* and *outports*; one or more processes may hold a single outport, while only one process can hold an inport.

Like NIL, Hermes supports the notion of *secure* programs. Programs are secure in the sense that the compiler ensures that variables are always used consistently; that is, a program cannot attempt to use a variable that is undefined. In addition, processes in the system can only affect each other through communication. The compiler uses a method called *typestate checking* to guarantee program security[105]. To promote safe programming, Hermes does not have pointers, but instead, provides *tables* for implementing dynamic data structures.

The original implementation of NIL consisted of a compiler that generated machine code directly from source code. However, the Hermes compiler generates target code using a machine independent, intermediate language called *LI* (for *Language of the Interpreter*) [14]. The compilers for both NIL and Hermes implement typestate checking. The LI is interpreted by a portable run-time system that handles process scheduling, interprocess communication, and memory management. Because Hermes programs are determined to be "safe" at compile time, the run-time system does not provide separate protection domains for each process. Allowing processes to execute in the same address space reduces context switch time and allows the run-time system to easily share data structures among processes. The run-time system uses Sun RPC [33] to allow processes on different machines to communicate with each other.

The Hermes run-time system is implemented on top of several versions of the UNIX operating system. Originally, the Hermes design team decided to implement the language using an interpreter rather than a compiler in order to decrease development time and increase portability. Developing the interpreter took much longer than expected. In retrospect, the designers concluded that the effort devoted to writing the interpreter could have been better spent on a compiler [14]. They also felt that UNIX standardization obviated their original concerns about portability.

2.3 Implementation Issues

For a programming language to be useful, there must exist tools for translating source code written in the language to a target architecture and allowing the translated code to execute. This section presents three important implementation issues: translation, portability, and run-time support.

2.3.1 Translation

The translation of source code to the target-machine instructions can come about by compilation or through an interpreter. A compiler translates source code to machine code or possibly to an intermediate language, which is then compiled into machine code.

An interpreter parses and translates source code to virtual machine instructions, which are then executed by a virtual machine. In most cases, interpreters suffer a performance penalty for a variety of reasons. For example, some interpreters must parse the source code of a program on every execution. Also, virtual instructions have inherit overhead because they must be decoded and executed in software. Most concurrent programming languages use compilers; therefore, this section focuses on the compilation method. However, many of the techniques for implementing compilers and interpreters are similar.

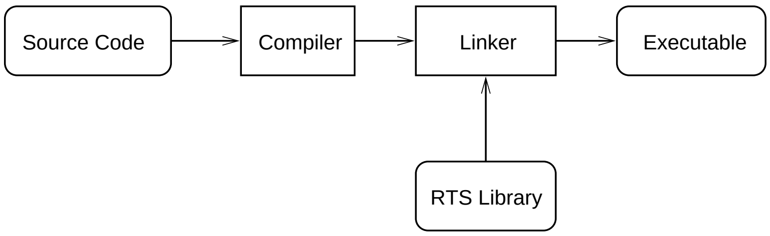

Figure 2.8: Compiling and Linking

A compiler consists of three basic phases: lexical analysis, parsing, and code generation. These phases are well understood and are thoroughly examined in [4]. After a compiler translates the source code, the resultant machine code is linked with a run-time system (RTS) library to produce an executable file for the target architecture and operating system. The linking phase will also resolve external references made by the source program(s) and adjust relative addresses. For example, a C program must be linked with standard libraries such as *libc* (the standard C library) and *libm* (the math library). In addition, every C program is linked with run-time startup code, usually called *crt0*. The steps to compile and link a program are illustrated in Figure 2.8.

The techniques used for lexing and parsing sequential languages apply easily to concurrent languages. However, code generation for concurrent programs introduces new complexities not found in sequential languages. The level of complexity depends on the programming model and the mechanisms used by the language. For many languages, the complexity of the code generation phase is reduced by implementing many language features in the run-time system. The run-time system provides an abstraction of the underlying operating system and processor. Using this method, the compiler can generate code that utilizes the run-time system for complex language mechanisms.

2.3.2 Run-Time Systems

Most concurrent programming languages require some amount of run-time support to provide language functionality during program execution. It is up to the run-time system to utilize the operating system interfaces efficiently. An obvious, but important point is that the run-time system must ensure that the execution of the compiled code adheres to the semantics of the language.

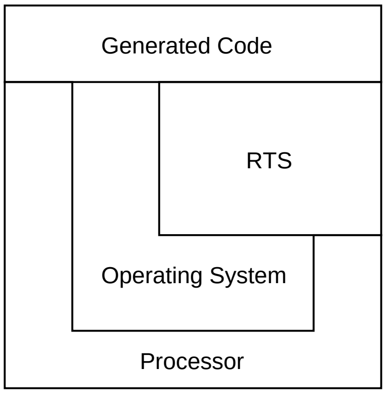

Figure 2.9: The Relationship Between the Generated Code and the RTS

Figure 2.9 illustrates the relationships among the generated code, the run-time system, the operating system, and the processor. The generated code consists of machine instructions, operating system calls, and run-time system calls. The amount of run-time system code will vary from language to language; for example, the run-time system for C is minimal, while the run-time system for a language that supports distributed shared memory is more substantial. In any case, the run-time system provides services to the generated code, which are implemented in terms of machine instructions and operating system calls. If constructed properly, the run-time system can hide the details of the underlying operating system from the generated code, allowing the generated code to be portable across operating systems.

Both sequential and concurrent languages usually provide run-time support for the following features:

- I/O (user input/output, file system access, and network access)
- Memory management

Both I/O and memory management are important to any programming language. A useful program usually needs access to user input or to data files, so most language implementations provide library support for user I/O (possibly through a graphical user interface) and file I/O. Network access is required by any program that wishes to have communicating processes residing on different machines or by a program that needs to utilize other network services (e.g., internet services). Although some sequential languages have library support for network access, all distributed languages have network support. However, the network support in a distributed language may be implicit rather than explicit, as is the case for sequential languages (e.g., sockets with C).

Most languages implement some form of memory management; it can range from the basic ability to explicitly allocate and deallocate regions of memory to fully automatic garbage collection. For instance, the C run-time library supplies the functions *malloc()* and *free()* for explicit memory allocation and deallocation, respectively. Most functional languages — such as Scheme [2] and ML [84] — and some imperative languages — such as Java, Ada 95, Icon [53], and Tcl [88] — have run-time support for implicit memory allocation and garbage collection. Traditionally, explicit memory allocation promotes efficient program execution, while implicit memory allocation results in safe program execution.

Unlike a sequential language, a run-time system for a concurrent programming language may provide services for one or more of the following features:

- Support for concurrency mechanisms
- Deadlock detection or prevention
- Thread migration
- Distributed shared memory

Paramount to any thread-oriented concurrent programming language is the support for concurrency mechanisms, including communication and synchronization primitives, as well as thread creation and destruction. For example, as explained in Section 2.1.4, a language that supports the message passing model must implement some form of the **send** and **receive** primitives. If a language supports asynchronous message passing, the language's run-time system might implement message queues or it might rely on the interprocess communication mechanisms supplied by the operating system. Likewise, a language that supports thread synchronization could simply rely on busy waiting or provide higher-level mechanisms such as semaphores or monitors. Busy waiting requires almost no run-time support, while semaphores and monitors require increasingly complex support. Finally, in a language that allows dynamic thread creation, the corresponding run-time system must provide means for allocating memory and starting a new thread of execution for each new thread. Similar to message passing support, the run-time system may rely on a thread library or operating system support for threads to implement basic thread functionality (see Section 2.4 for a discussion of thread systems).

Although support for concurrency mechanisms is central to most run-time systems for concurrent programming languages, a run-time system might also provide higher-level functionality. Deadlock is a common problem that arises in concurrent programs when there is a cyclic dependency involving two or more threads. A sophisticated run-time system can potentially detect and even prevent a deadlock situation. Another higher-level run-time system service is thread migration. A run-time system may decide to transfer a thread running on one machine to another, possibly to increase program efficiency by moving a thread to a lightly loaded machine or simply to keep the program running in the event that the current host machine needs to be shutdown.

If a concurrent programming language supports the shared address space model for computation, the run-time system must implement some form of distributed shared memory (DSM). As described in Section 2.1.6, DSM requires the run-time system to ensure that the shared memory is always in a consistent state. To accomplish data coherency, a run-time system will implement one of many memory consistency protocols. Such protocols are usually expensive in terms of performance, thus they are still a subject of research [71, 47, 106, 120].

2.3.3 Portability

Portability is a complicated software engineering problem that is handled in different ways by different types of programs, from applications to compilers to operating systems. For example, the GNU C compiler uses a register transfer language to simplify retargeting the back-end to different architectures.

In many ways, a run-time system is similar to an operating system kernel. A run-time system provides a set of interfaces to basic services. These interfaces are an abstraction of the underlying operating system and architecture. The run-time system interfaces are used by compiler-generated code or a language interpreter.

Many programs can achieve reasonable portability by conforming to ANSI C and using POSIX library functions. However, concurrent programming languages make extensive use of threads and communication interfaces. Unfortunately, the interfaces for threads and communication can vary greatly from platform to platform, making it difficult to implement a run-time system for multiple targets. This is true despite standardization efforts such as MPI [110] and Pthreads [63] (see Section 2.4). It is also not clear if the standard interfaces are always desirable. For example, it has been observed that many current thread interfaces, including Pthreads, are not very flexible [54].

Portability can be viewed from two perspectives: from *portability clients* (programs for which portability is desired or required) and from *portability providers* (programs, libraries, or interfaces designed to provide portability). It is infeasible for portability clients to completely rely on portability providers, as a particular provider usually only targets a subset of possible platforms. To this end, clients must take measures to ensure that multiple providers can be utilized. For example, the GNU Autoconf tool [81] aids portability clients in assessing the capabilities of a target platform so appropriate, automatic adjustments can be made to the source code. Such a tool is necessary to effectively target a large number of different platforms; otherwise, manual techniques result in an unwieldy number of conditional compilation macros, making code unmanageable. We generally take the point of view of the language implementation — i.e., a portability client — and are concerned with techniques that allow a language to be implemented as quickly as possible with good performance for a large number of target platforms.

Tools can help manage portability to some extent, but (especially in the domain of run-time systems) proper structuring can also result in good portability. The next section discusses this issue further.

2.3.4 Run-Time System Structure

This section describes three basic approaches to structuring a run-time system: monolithic, virtual machine, and component based. The latter two approaches lead to more portable run-time systems.

2.3.4.1 Monolithic Approach

The monolithic approach, illustrated in Figure 2.10, is a straightforward method of directly implementing a run-time system in terms of the interfaces provided by the underlying operating system. This approach does not offer much in terms of portability. For example, the original SR run-time system was implemented using standard UNIX interfaces (see Section 3.2).

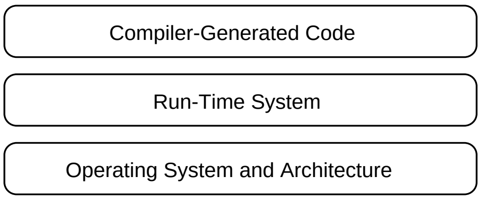

Figure 2.10: The Monolithic Approach

2.3.4.2 Virtual Machine Approach

Some efforts aim to provide portable run-time support either by utilizing standardized interfaces or by providing a portable *virtual machine* (see Figure 2.11). The virtual machine is simply a layer between the run-time system and the kernel primitives or the hardware. This layer provides an abstraction of the underlying operating system and architecture.

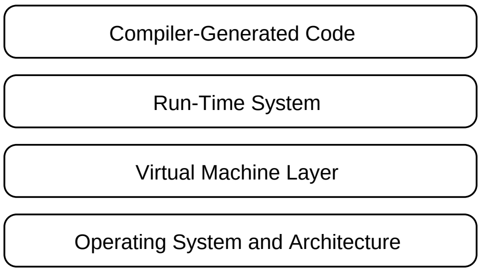

Figure 2.11: The Virtual Machine Approach

The Panda system [29] is a portable platform for supporting parallel programming languages, which consists of two layers: the system layer and the Panda layer. The system layer interfaces with operating system kernel primitives and is kernel dependent. The Panda layer provides a virtual machine for supporting run-time systems, including threads, message passing, remote procedure call, totally-ordered group communication, and collective communication. Panda targets run-time systems that provide distributed shared data or distributed shared objects.

Nexus [49] is a distributed thread library that provides a virtual machine layer similar to Panda. Nexus was designed to be used as a back-end for concurrent programming languages, e.g., CC++ and Fortran-M [48]. It provides support for creating threads on a processor, for specifying synchronization among the threads of a node, and for communication among threads through the active-message paradigm.

2.3.4.3 Component Approach

The component approach, illustrated in Figure 2.12, attempts to separate run-time system functionality and system-dependent interfaces into individual modules[1]. Various forms of the component approach have been proposed in [25] and [68]. The basic idea is to separate system dependencies into components so that they can be specialized on a per-platform basis. Note that this approach can be complementary to the virtual machine approach; i.e., system-dependent components may be implemented in terms of virtual machine interfaces for a particular platform. However, from the perspective of the language implementation, there is no "bottom" layer unless a particular implementation chooses to use such a layer.

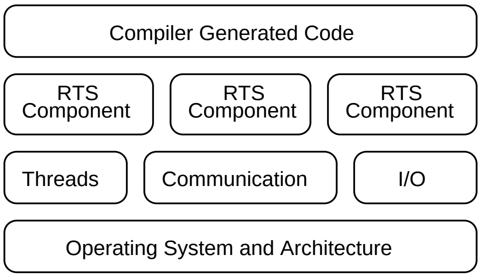

Figure 2.12: The Component Approach

A component is a very general entity that encapsulates some type of functionality. We do not place any constraints on components; for example, a component can be

[1]Figure 2.12 depicts only two levels of components, but it is feasible to have arbitrary hierarchies of components.

implemented as an object-oriented class. However, due to performance requirements it is often necessary to avoid traditional black-box techniques [72]. As a consequence, some components may have exposed state that can be manipulated by other components.

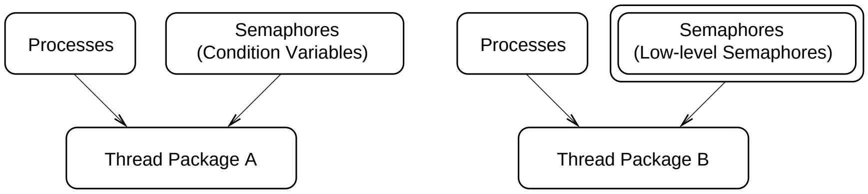

Figure 2.13: Component Specialization

As a specific example, consider a run-time system that supports processes and semaphores (see Figure 2.13). For some implementations, processes and semaphores may be implemented in terms of the low-level interfaces provided by a thread package. In particular, processes may be implemented as threads and semaphores may be implemented with condition variables. These basic thread primitives exist in many thread packages. However, a more advanced thread package or operating system kernel may provide efficient semaphores. The right-hand side of Figure 2.13 illustrates this specialization. Although this example is simple enough to be handled well by either a virtual machine layer or by conditional compilation, it also illustrates the kind of flexibility that the component approach can provide.

Although the component approach facilitates specialization of components, it is desirable to minimize the number of specialized components (because more specialized components means less portable run-time systems). The idea is that performance optimizations through specialization will incur a cost in portability. In the component approach, portability can be achieved locally, that is, within individual components.

The component approach is still an active research topic. Many issues still need to be resolved before such an approach can be generally useful. An important concern is the determination and/or specification of dependencies among components. A very straightforward approach is to clearly document dependencies and any state that is shared among components; when a component is specialized, all the information needed for replacement is provided. This approach is being used successfully in the Flux Operating System Toolkit [44], a toolkit for building kernels.

Another crucial concern is the mitigation of specialization costs. We foresee a technique and corresponding tool that helps track components and their compositions. A good model for such a technique is the x-kernel [62], which is an architecture for specifying and composing micro-protocols. While the domain of the x-kernel is constrained to communication protocols, it demonstrates how different functional components can be combined to produce high-performance systems.

2.3.4.4 A Comparison of Approaches

It is important to understand the differences between the virtual machine approach and the component approach. The virtual machine approach presents a fixed interface to a run-time system. Such an interface results in less complexity at the platform independent run-time system layer and is the cornerstone for portability. In addition, virtual machines can be designed to be extremely modular "underneath" the fixed interface. Such modularity results in virtual machines that are easier to port (e.g., both Panda [29] and Nexus [49] have modular designs). However, the fixed interface can hide system dependent details that may be important to the implementation of a run-time system. If a language implementor "codes around" the fixed interface, the advantages of the virtual machine are lost.

The component approach offers greater flexibility in specializing performance critical portions of a run-time system. In addition, depending on the particular virtual machine, the component approach may lead to quicker development time for new platforms because only the low-level functionality that is required by the run-time system is implemented. This is true when compared to virtual machines, such as Nexus [49], that support different abstractions, some of which may not be needed by a particular run-time system. However, the component approach, at least initially, may result in more complex run-time systems due to the flexibility of specialization.

2.4 Thread Systems

As introduced in Section 2.1, the concept of a thread is fundamental to concurrent programming. Most of the languages presented in Section 2.2 provide language-level support for threads and thread synchronization. Section 2.3.2 mentions that it is up to a run-time system to provide the low-level implementation of threads. It is not surprising that the way threads are implemented in a run-time system has a significant impact on the performance of language-level threads and the programs that use them. Some of the main results of this dissertation are based on analyzing the ways in which run-time systems implement threads. A run-time system implementor usually chooses between writing a custom thread system or using an existing thread system. Most language implementations use the latter approach. This section describes thread system interfaces and implementation alternatives.

Most processor architectures do not have direct support for threads. Therefore, a thread has traditionally been a software entity and is implemented by libraries, operating systems, or a combination of the two. A thread system provides interfaces for basic thread manipulation, including thread creation, termination, synchronization, and identification. These interfaces allow a program written in a sequential programming language to have multiple threads of control. Most thread systems are implemented as C libraries. Although some thread systems are provided as a C++ class, these usually are built on top of a lower-level C library for threads.

While many different thread interfaces exist, most have converged to the IEEE Pthreads standard [64]. Libraries based on this standard are the most common across different operating systems. Because the Pthreads interface is so common, we will refer to any thread system that provides an interface that is similar to the Pthreads interface as a *Pthreads-style* thread system. We will present some of the common thread interfaces in terms of the Pthreads standard.

2.4.1 Basic Concepts

In a C program augmented with calls to a thread library, each thread has its own stack and register context (including its own instruction pointer). All the threads can access the same global variables and any local variables made accessible through pointers. Generally, threads can also share resources provided by the operating system. For example, all the threads in a UNIX process can share the file descriptors associated with the process. Thus, threads can share access to devices opened as file descriptors, such as network connections.

2.4.1.1 Thread Creation and Termination

Thread creation and termination are accomplished with the following functions:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
       void * (*start_routine)(void *), void *arg);

void pthread_exit(void *retval);
```

The function `pthread_create` is used to start a new thread. On completion, a new stack and internal data structures (in the thread library, operating system, or both) are created on behalf of the new thread. The thread will execute the function pointed to by `start_routine`, which is passed the value of `arg`. A thread exits when it returns from `start_routine` or when it calls `pthread_exit`. The latest Pthreads standard also allows threads to be terminated asynchronously. That is, one thread can kill another thread. However, using asynchronous cancellation is considered unsafe except in rare situations.

2.4.1.2 Mutexes

Mutual exclusion to critical sections in a program is accomplished with the following functions on mutex variables:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The `pthread_mutex_lock` function is used to gain entry to a critical section and `pthread_mutex_unlock` to announce exit from a critical section. If a thread attempts to acquire a lock that is already held by another thread, it will block until the second thread releases the lock. Most Pthread implementations use a queue to block threads waiting on a mutex. Mutex locks should generally be held for only short periods of time. Also, the thread that locks a mutex variable must also be the thread that unlocks the same variable.

2.4.1.3 Condition Variables

Pthread condition variables provide a form of condition synchronization among threads. The condition variable represents a queue of waiting threads on a particular condition. The four most common functions used with condition variables are:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

The `pthread_cond_wait` function allows a thread to wait for a signal from another process. The signal usually indicates that the condition associated with the condition variable is now true. The `pthread_cond_timedwait` function is similar to the wait function except that the thread will only wait until `abstime`. The return value of `pthread_cond_timedwait` indicates if a signal was received or if the function returned because `abstime` had been reached. A programmer uses `pthread_cond_signal` to wake up another thread waiting on the specified condition variable and `pthread_cond_broadcast` to wake up all the threads waiting on the specified condition variable. Signals sent to condition variables with no waiting threads are lost. Condition variables combined with mutexes give the C programmer a rudimentary form of monitors (as described in Section 2.1.3).

2.4.1.4 Thread-specific Data

Pthreads provides an interface for creating global names that can have different values for each thread. These values are called thread-specific data and they are accessed with the following functions:

```
int pthread_setspecific(pthread_key_t key, const void *value);

void *pthread_getspecific(pthread_key_t key);
```

The `key` is used as the name of the data represented by `value`. The `pthread_getspecific` function returns the value associated with a particular key for the calling thread. The functions for thread-specific data have several uses. Often, thread-specific data are used to implement per-thread `errno` values and to make C functions that require *static* data thread safe. In addition, as we will see in Chapter 5, thread-specific data are useful in the implementation of concurrent run-time systems.

2.4.1.5 Other Interfaces

In addition to the functions listed above, the Pthreads standard specifies other interfaces, including ones for semaphores, reader-writer locks, and thread cancellation. Most Pthreads objects have support for dynamic creation and destruction. The Pthreads interface also specifies attributes for most objects. These attributes can change the behavior of an object to suit the needs of an application. Some of the attributes that can be modified include the thread stack size and scheduling policy. See the standard for further information on the entire Pthread interface [64].

2.4.1.6 Example: Implementing Semaphores with Pthreads

A Pthreads implementation of semaphores is listed in Figure 2.14. A semaphore is a struct with four elements: a value, a blocked count, a mutex lock, and a condition variable. The mutex variable and condition variable are used to make a simple monitor for the semaphore. The `value` variable is used to keep track of the number of `V` operations and the `blocked` variable is used to keep track of the number of threads waiting for the value of the semaphore to increase.

2.4.2 Implementations

The implementation of a Pthreads-style interface is responsible for managing and scheduling threads. Most of the functions involve some form of queue manipulation. This section gives an overview of the major implementation techniques. A more detailed analysis of thread implementation and incorporating threads into run-time systems is presented in Chapter 5.

2.4.2.1 User-level

A common approach to implementing a thread system is to do so entirely in user-space without kernel-support. Because the kernel is not involved, user-level implementations have very efficient thread operations. For this reason, user-level threads are often referred to as “light-weight” threads. Two notable problems with user-level threads are the lack of blocking I/O support and the mismatch between the kernel scheduler and the user-level

```
#include<pthread.h>

typedef struct semaphore {
        int value;
        int blocked;
        pthread_mutex_t lock;
        pthread_cond_t cv;
} semaphore_t;

void sem_P(semaphore_t *sem)
{
        pthread_mutex_lock(&(sem->lock));
        if (sem->value >= 0) {
                sem->value--;
        } else {
                sem->blocked++;
                while (sem->value == 0) {
                        pthread_cond_wait(&(sem->cv), &(sem->lock));
                }
        }
        pthread_mutex_unlock(&(sem->lock));
}

void sem_V(semaphore_t *sem)
{
        pthread_mutex_lock(&(sem->lock));
        if (sem->blocked == 0) {
                sem->value++;
        } else {
                sem->blocked--;
                pthread_cond_signal(&(sem->cv));
        }
        pthread_mutex_unlock(&(sem->lock));
}
```

Figure 2.14: A Pthreads Semaphore Implementation

scheduler. The former problem requires sophisticated support to allow user-level threads to block on I/O operations. The latter problem can lead to poor performance on shared-memory multiprocessors if the kernel preempts a user-level thread that is holding a lock needed by other threads. In addition, user-level thread systems are not usually integrated with the standard C library functions. Some key C library functions such as `malloc()`, `free()`, and those for buffered I/O (`fopen()`, etc.) manipulate global variables; thus, multiple, simultaneous calls to any one of these functions can introduce race conditions on the global data. Either the programmer has to take care when using these functions in a multi-threaded program or the functions must be rewritten to avoid race conditions.

2.4.2.2 Kernel-level

Another approach to thread implementation is to have threads managed by the operating system kernel. Operating systems such as Linux, Solaris, and Windows NT all provide kernel-level threads. The advantage of kernel threads is that threads can easily perform blocking I/O operations and the kernel can schedule threads onto different processors on a shared memory multiprocessor. Unfortunately, kernel threads are usually very expensive to create, synchronize, and switch between due to the high cost of crossing the user/kernel boundary on every thread operation.

2.4.2.3 Hybrid

In a hybrid implementation, the thread system multiplexes user-level threads onto kernel-level threads in an attempt to get the benefits of both approaches. This is the approach taken in Solaris Threads [91]. In addition, more advanced techniques such as Scheduler Activations [7], First Class User-Level Threads [82], and, more recently, Strands [101] have been used to enable better coordination between user-level scheduling and kernel scheduling. Unfortunately, all these techniques require kernel support and are currently restricted to research systems or private kernel interfaces [2].

[2]For example, Solaris now has support for scheduler activations, but the kernel interfaces are not publicly available.

2.4.3 Examples

2.4.3.1 OpenThreads

OpenThreads [54] is an example of a user-level thread package that provides a Pthreads-style interface. In addition to basic thread operations, OpenThreads provides a meta-interface that can be used to modify the behavior of the basic thread abstractions. Among other things, the meta-interface allows a client to modify the stack allocation function, to customize the queueing mechanism for thread scheduling, and to add callback functions to various thread events for tracing and debugging. OpenThreads is a good example of a thread package that allows a reasonable degree of customization.

OpenThreads is implemented with QuickThreads [70], a toolkit for building user-level thread systems. Currently, the implementation does not provide multiprocessor support and, like many user-level thread systems, it lacks good integration with the standard C library.

Despite its shortcomings, OpenThreads is quite useful. It is a very fast user-level thread system that can be used as a Pthreads replacement. For example, OpenThreads has been used to implement threads in the Panda portability layer [55].

2.4.3.2 Linux Threads

The Linux kernel supports threads with the `__clone()` system call. This system call creates a new process that shares everything accessible in the original process, including the address space, file descriptors, and virtual memory maps. However, the new process is given a new process control block and its own stack. The `__clone()` system call is similar to the `sproc()` system call found in IRIX.

LinuxThreads [75] is a Pthreads compatible thread system built on top of the `__clone()` system call. Because threads are scheduled by the kernel, LinuxThreads supports blocking I/O operations and multiprocessors. However, each thread is really a Linux process, so the number of threads a program can have is limited to the total number of processes allowed by the kernel (which is usually a fixed value around 256 or 512). The Linux kernel does not provide system calls for thread synchronization; therefore, the LinuxThreads library provides additional code to support operations on mutex and condition variables (using pipes to block threads). The library code is complex, so although LinuxThreads are kernel-level threads, full Pthreads functionality requires a significant amount of user-level support.

2.4.3.3 Solaris Threads

Solaris Threads [91] employs a hybrid approach to the implementation of threads in an attempt to achieve the advantages of both user-level and kernel-level threads. Similar to Linux, Solaris provides system calls to create kernel-level threads. These threads are called *light-weight processes* (LWPs). The Solaris Threads library multiplexes user-level threads onto LWPs. The idea is to create new LWPs as a program requires more parallelism or if a user-level thread blocks on I/O. The implementation is rather complex, thus adding overhead to basic user-level thread operations and making them less efficient than a user-level only thread library.

2.5 Summary

This chapter has reviewed various aspects of concurrent programming and concurrent programming languages, including language mechanisms, language examples, implementation issues, and thread systems. A central point raised in this chapter is that different languages support different concurrency mechanisms and these mechanisms are implemented with the help of a run-time system. As suggested in Section 2.2, the form and requirements of a run-time system vary greatly depending on the language and the underlying operating system

Different languages have different goals for and approaches to concurrency. These goals and approaches are reflected in each language's semantics and implementation. For example, Java provides synchronized objects. This approach is sufficient for many types of programs, but the programmer has little control over the synchronization policies, requiring complex work-around code in some cases. In addition, the language definition is loose on its specification of thread scheduling, so an implementation has quite a bit of flexibility in its scheduling policy. Unlike in Java where the language constructs for concurrency augment the core language constructs, Ada 95 provides an entirely separate set of language constructs for specifying concurrency. Thus, Ada sacrifices simplicity, but does not burden sequential constructs with synchronization concerns (reflecting Ada's general purpose design). In Modula-3, threads are provided as an external module and offer a very simple interface based on mutex and condition variables. Lynx provides message passing between different programs, thus supporting distributed programming. Coroutines are used in the Lynx implementation to provide pseudo-parallelism within single program. Orca is also designed for distributed programming, but the central construct is the shared data-object. Finally, Hermes supports distributed programming by integrating processes and message passing into the language.

While the languages presented in this chapter differ with respect to their concurrency semantics, their implementations are similar with respect to portability. Most implementations of these languages employ ad hoc approaches or standard software layering techniques. Some implementations rely on standard interfaces for portability; for example, Ada 95 and Modula-3 both have implementations that rely on Pthreads. Also, Hermes assumed UNIX as the underlying operating system.

Central to many concurrent run-time systems is a thread implementation. Many languages rely on a thread system rather than implementing threads directly. Today, most thread systems provide a Pthreads-style interface, which allows a programmer to create, destroy, and synchronize threads. However, the Pthreads-style interface can be implemented in different ways. The three broad implementation alternatives are user-level, kernel-level, and hybrid approaches. We will find in Chapter 5 that the Pthreads-style interface poses some fundamental limitations in the context of concurrent run-time systems.

Chapter 3

The SR Language and Implementation

SR is a concurrent programming language based on sequential processes that interact using multiple communication mechanisms [11, 10]. Through orthogonal use of SR's basic communication primitives, a programmer can employ asynchronous message passing, synchronous message passing, process creation, remote procedure call, or rendezvous. SR is expressive enough to write sophisticated applications, yet it is simple enough to make it easy to learn.

The programming model supported by SR strikes a balance between efficient low-level primitives and expensive high-level primitives. For example, SR allows shared variables on a single node, but it does not provide shared memory across machines. Therefore, SR can take advantage of multiprocessors and networked computers, but it does not support distributed shared memory, which is currently very expensive to implement.

SR's multiple communication mechanisms and relatively low-level concurrent programming model make it ideal for system-level and user-level programming. This chapter briefly describes the SR language and its current implementation. This description is important to the development of later chapters as much of the research in this dissertation is based on SR and its original implementation[1]. A comprehensive description of SR is given in [11] and [10].

[1]By "original implementation", we mean the implementation supported and distributed by the University of Arizona.

3.1 Language Overview

SR was designed and implemented from the ground up. Therefore, its mechanisms for parallelism, communication, and synchronization are tightly integrated into the language. The concurrent aspects of the language are based on just a few, simple constructs. This section summarizes the major language components, including the sequential aspects, resources, operations, and some additional language mechanisms. In addition, this section contains some program examples.

3.1.1 Sequential Aspects

In some respects, SR resembles other imperative languages like C, Modula-2, and Java. For instance, SR supports basic types such as booleans, integers, reals, characters, and strings. SR also supports enumerations, records, and unions. These types can be used to form more complex user-defined types. Similar to other languages, SR supports pointers and dynamic storage allocation. SR is strongly typed and performs almost all type checking at compile time. However, some type checking, such as array bounds checking, is done at run time.

SR provides several constructs for sequential control. Unlike many languages, the **if** and **do** statements are based on nondeterministic selection introduced in CSP [39]. For example, the **if** statement can have several arms (also known as guarded commands), but the order in which the guards are evaluated is not specified. Likewise, the **do** statement, which is used for indefinite iteration, may have one or more guarded commands whose guards are evaluated nondeterministically until all of them evaluate to false. If a guard evaluates to true, its command is executed and the execution of the **do** statement is repeated. SR provides the **fa** (for all) statement for definite iteration.

Although SR provides a **procedure** declaration for specifying procedures and functions, **procedure** is really an abbreviation for an operation declaration and a **proc**. Operations and procs are described further in Section 3.1.3.

3.1.2 Resources

A *resource* encapsulates data and code that implement a particular concept, algorithm, or abstract type. Resources can be created and destroyed using the **create** and **destroy** statements, respectively. A resource serves as a template that can be dynamically instantiated. Upon creation, resources can be passed parameters, which can be used for initialization (e.g., a stack resource may require a size parameter). Each resource instance contains its own local data and execution state. A typical SR program consists of several interacting resources.

In general, a resource is composed of two parts: the *spec*, which specifies the resource interface, and the *body*, which contains the code that implements the resource. The general form for a resource is shown in Figure 3.1.

resource *resource_name*
  imports
  constant, type, or operation declarations
body *resource_name (parameters)*
  imports
  declarations, statements, procs
  final code
end *resource_name*

Figure 3.1: The General Form of an SR Resource

The spec and body of a resource can be split into two parts, which either reside in the same file or are placed into individual files to support separate compilation. Thus, the implementation (i.e., the body) of a resource can change while the interface (i.e., the spec) remains the same.

The spec portion specifies the interface to the resource; all objects declared in the spec are visible to other resources that import this resource. A resource spec can import other resources; in this case, all imported objects are made available to any other resource that imports this resource. In addition to specifying imports, the spec can also contain declarations for constants, types, and operations. SR constants and types are similar to those found in other imperative languages, while operations define the entry points into a resource.

Like the spec, the body can import other resources as well as declare constants, types, and operations. However, the body also consists of variable declarations, program statements, proc definitions, and final code. *Procs* are bodies of code that service operations;

in their most basic form, procs can be used just like procedures or functions found in other languages. Section 3.1.3 describes an additional way to invoke procs to perform dynamic process creation. The body of a resource can also specify *final code* that is executed just before the resource instance is destroyed; this allows a resource to finish a computation or clean up allocated memory. Resources are destroyed explicitly with the **destroy** statement.

In order to facilitate the creation and use of resources, SR provides a *resource capability* type. When a resource is created with the **create** statement, a resource capability is returned. This return value can be used as a handle to the newly created resource or it can be ignored. Operations defined by the created resource can be accessed through its resource capability.

A construct similar to the resource is the *global*. Globals are defined just like resources except the **global** keyword is used instead of the **resource** keyword and they are not parameterized. Unlike resources, the spec part of a global may contain variable declarations. An SR program may contain several globals. In order to access the objects in a global, a resource must import the global. Unlike resources, only one instance of each global is created at run time on a virtual machine (see Section 3.1.4). A global component allows two or more resources to share data, constants, types, and operations.

3.1.3 Operations

An operation is an abstraction of a procedure call that can designate code local to a resource or an entry point into a resource. Operations can be invoked using the **call** or **send** statements, while operations can be serviced with a **proc** or the **in** statement. Function invocation has the same effect as the **call** statement, except that a return value is expected. How an operation is invoked combined with the way it is serviced determines the method of communication. The following table lists the possibilities.

Invocation	Service	Effect
call	**proc**	procedure call (possibly remote or recursive)
call	**in**	rendezvous
send	**proc**	dynamic process creation
send	**in**	asynchronous message passing

The **in** statement is a powerful mechanism for servicing operation invocations. In its most general form, the **in** statement allows one or more arms that specify operation commands. An operation command specifies an operation, an optional synchronization expression, an optional scheduling expression, and a statement block. Specifying an operation allows it to be serviced by the **in** statement. A synchronization expression is a boolean expression possibly involving variables or arguments from the operation invocation; it must evaluate to true before the invocation will be serviced, in which case the invocation is considered selectable. A scheduling expression determines the order in which selectable invocations are serviced. If a scheduling expression is omitted, then selectable invocations are serviced in FCFS (first-come, first-served) order. In addition, if there are selectable invocations for two or more operations specified by the **in** statement, they are also serviced in FCFS order. If an invocation is accepted by the **in** statement, the statement block corresponding to the invoked operation will be executed. An **in** statement may contain a special purpose operation command, **else**, followed by a statement block, which is executed if no invocation is selectable.

Local operations represent code that is local to a resource, either in the form of a proc or in the arm of an input statement. Generally, local operations are only visible to the defining resource and it is possible for the compiler to generate optimized code for such operations. However, local operations can be made visible to other resources using operations capabilities (see below). Local operations can also be created dynamically.

SR provides *operation capabilities*, which serve as pointers to operations. Operation capabilities can be declared as variables, assigned to, passed as parameters, and used in invocation statements. The invocation of an operation capability has the same effect as invoking the operation to which it points. SR's operation capability is a powerful language feature: It enables programs to dynamically create arbitrarily complex communication paths.

3.1.4 Virtual Machines

SR resources exist in the context of a *virtual machine.* A virtual machine essentially encapsulates a single address space. When an SR program is executed, a virtual machine is created on the local computer. An SR program can explicitly create additional virtual machines on the local computer or on remote computers. When a virtual machine is created with the **create** statement, a *virtual machine capability* is returned. Virtual machine capabilities are analogous to operation and resource capabilities. It is not possible for two virtual machines to share memory; thus, distributed programs must communication through some form of message passing. Figure 3.2 illustrates the relationships between virtual machines, resources, and operations.

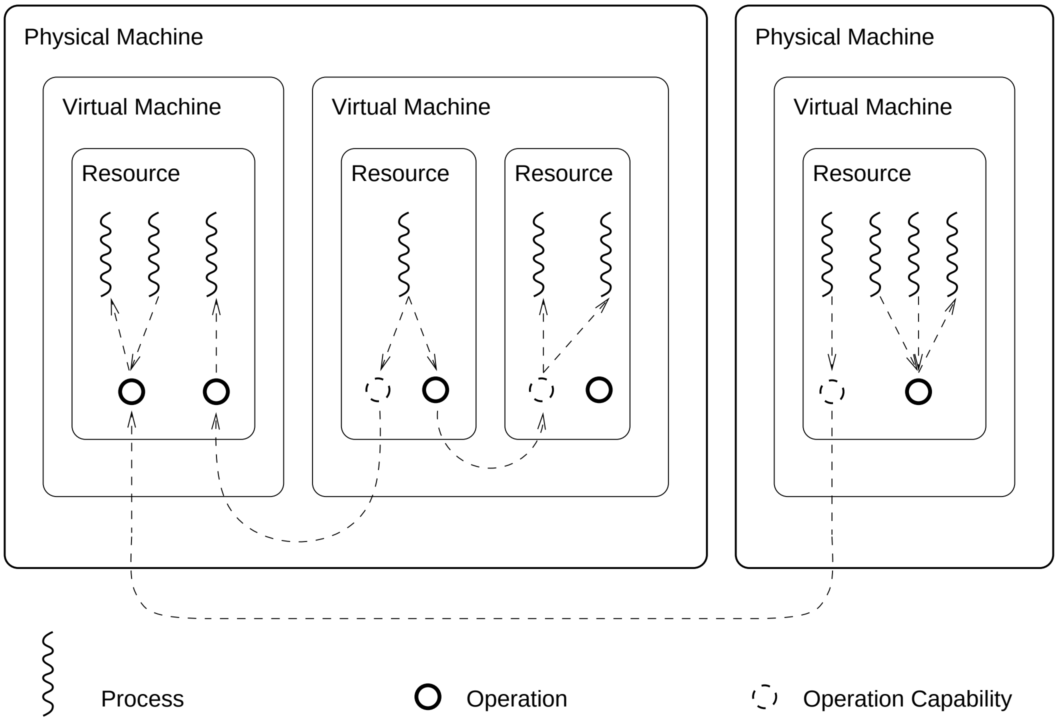

Figure 3.2: Virtual Machines and Resources

3.1.5 Additional Language Mechanisms

Process SR provides the **process** declaration to simplify the creation of simple processes. A process declaration is really an abbreviation for an operation declaration, a proc definition, and a send to the proc. A quantifier can be used with the process declaration to create several similar processes that differ only in values of one or more quantified variables.

Receive The **receive** statement is a short form of the **in** statement that blocks for a single invocation on a single operation without synchronization or scheduling expressions. This statement can be used for simple message passing.

Concurrent Invocation As indicated in Section 3.1.3, issuing a **send** statement on an operation serviced by a **proc** will cause a new process to be created. Parallel execution can also be accomplished with concurrent invocations using the **co** statement. The **co** statement allows a program to simultaneously invoke an arbitrary number of operations. Different forms of the **co** statement allow qualifiers to determine how many concurrent commands are executed and allow post-processing code for each concurrent command to be specified.

Semaphores SR also provides a *semaphore* type for synchronizing the execution of concurrent processes (see Section 2.1.2). An SR program can declare individual semaphores as well as arrays of semaphores. The keywords **P** and **V** perform the usual operations on a specified semaphore. The SR compiler treats semaphores as operations with no arguments; a **P** statement is treated as a **receive** statement and a **V** statement is treated as a **send** statement. For simple semaphores and certain uses of operations, the compiler can generate optimized code that directly uses low-level semaphores instead of using **send** and **receive**. (See Section 3.2.2 for a discussion of the implementation of semaphores in the run-time system.)

Input/Output The original SR implementation, which is UNIX-based, provides several mechanisms for interacting with input/output devices and files. In particular, SR offers mechanisms for reading user input, manipulating files, and reading arguments from the command line. The SR language definition also includes *external operations*, which can be written in any language that supports the C calling conventions.

3.1.6 Example Programs

This section presents two programs that illustrate the utility of the SR language. Unfortunately, because SR is a very expressive language, it is difficult to exhibit all of the language mechanisms in only two programs. Nevertheless, these two programs show key aspects of the language. The first program implements a stack resource using remote procedure call; the second program implements a centralized solution to the dining philosophers problem using rendezvous. These programs are taken from [10].

3.1.6.1 A Stack Resource

The `Stack` resource listed in Figure 3.3 is a relatively simply SR program that defines an abstract type. The spec portion of the resource declares a result type and two operations: `push` and `pop`. Any resource that imports `Stack` will have access to `result`, `push`, and `pop`. The body specifies that `Stack` is a parameterized resource; the creator of the `Stack` resource must specify the size of the stack. The body of `Stack` also declares two variables local to the resource: `store` and `top`. In addition, the `push` and `pop` procs are declared to service the corresponding operations declared in the spec; these operations perform the usual stack functions.

The `Stack_User` resource listed in Figure 3.4 illustrates use of the `Stack` resource. Although the `Stack` resource is a perfectly good sequential entity, it can also be used in a distributed context. That is, it is possible for concurrently executing processes to access the same stack using remote procedure call semantics. However, since the `store` and `top` are shared variables, access to them would need to be synchronized. By using the **in** statement instead of procs to service the stack operations (push and pop), this conflict could be easily avoided. For a more detailed explanation of this example, see [10]. The next example program, dining philosophers, shows how to use the **in** statement for both communication and synchronization.

```
resource Stack
  type result = enum(OK, OVERFLOW, UNDERFLOW)
  op push(item: int) returns r: result
  op pop(res item: int) returns r: result
body Stack(size: int)
  var store[1:size]: int, top: int := 0
  proc push(item) returns r
      if top < size → store[++top] := item; r := OK
      [] top = size → r := OVERFLOW
      fi
  end
  proc pop(item) returns r
      if top > 0 → item := store[top−−]; r := OK
      [] top = 0 → r := UNDERFLOW
      fi
  end
end Stack
```

Figure 3.3: A Stack Resource

```
resource Stack_User()
  import Stack
  var x: Stack.result
  var s1, s2: cap Stack
  var y: int
  s1 := create Stack(10); s2 := create Stack(20)
     ...
  s1.push(4); s1.push(37); s2.push(98)
  if (x := s1.pop(y)) ≠ OK → ... fi
  if (x := s2.pop(y)) ≠ OK → ... fi
     ...
end
```

Figure 3.4: A Stack Resource Client

3.1.6.2 Dining Philosophers

This centralized solution to the dining philosophers problem listed in Figures 3.6, 3.7, and 3.8 demonstrates the use of rendezvous in SR. The basic structure of this solution is illustrated in Figure 3.5. Each philosopher is represented by an SR process. The philosophers coordinate their actions (acquiring and releasing forks) through a servant process. It is up to the servant to maintain the *philosopher invariant*: a philosopher must have both a left fork and a right fork to eat and each fork may be held by only one philosopher at a time. The servant keeps track of which philosophers are eating and uses this information to maintain the invariant. Thus, the servant determines if new philosophers may acquire their forks and eat or if they must wait for other philosophers to finish eating.

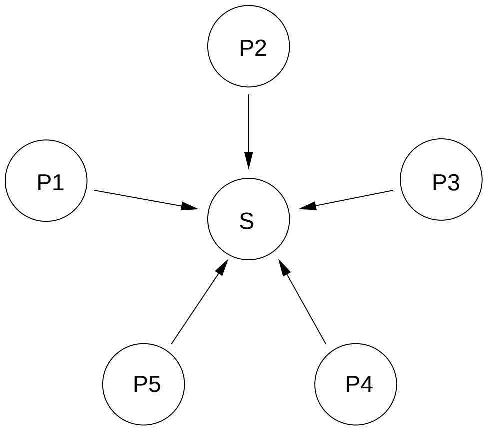

Figure 3.5: Structure of the Centralized Solution to the Dining Philosophers Problem

The heart of this solution lies in the `Servant` resource listed in Figure 3.6. On creation of the `Servant` resource, a single process is invoked. This process executes an infinite loop, waiting for requests using the **in** statement. Each arm of the **in** statement specifies a different operation. One arm services invocations on the `getforks` operation and the other arm service invocations on the `relforks` operation. Because the **in** statement allows invocations to be conditionally accepted, it is easy to maintain the philosopher invariant. The arm for `getforks` uses a synchronization expression to determine if a philosopher may eat or not. Only invocations that satisfy the synchronization expression will be serviced.

The `Philosopher` resource listed in Figure 3.7 contains code to start a philosopher process. The philosopher process executes a loop that iterates `t` times. In each loop iteration, the philosopher eats (acquires the forks), then thinks (releases the forks).

```
resource Servant
  op getforks(id : int)  # called by Philosophers
  op relforks(id : int)
body Servant(n : int)
  process server
    var eating[1:n] := ([n] false)
    do true →
      in getforks(id) st not eating[(id mod n) + 1]
          and not eating[((id − 2) mod n) + 1] →
              eating[id] := true
      [] relforks(id) →
              eating[id] := false
      ni
    od
  end
end
```

Figure 3.6: The Dining Philosopher's Servant Resource

```
resource Philosopher
  import Servant
body Philosopher(s : cap Servant; id, t : int)
  process phil
    fa i := 1 to t →
      s.getforks(id)
      write("Philosopher", id, "is eating")
      s.relforks(id)
      write("Philosopher", id, "is thinking")
    af
  end
end
```

Figure 3.7: The Dining Philosopher's Philosopher Resource

Execution of the dining philosophers program begins with the `Main` resource listed in Figure 3.8. A process is created to execute the body of `Main`. This process queries the user for the number of philosophers and the number of iterations each philosopher should execute.

The rest of the code in `Main` creates a `Servant` resource and the specified number of `Philosopher` resources. The `Servant` resource is created on the current virtual machine (i.e., the virtual machine on which `Main` is executing). However, each `Philosopher` resource is created on a new virtual machine. The `Main` resource identifies three hosts: ivy, sleep, and elysium. The user-defined number of philosophers are evenly distributed among the three machines. Notice the **create** statement is used to create a virtual machine in the same way a resource is created.

```
resource Main()
  import Philosopher, Servant
  var n, t: int
  writes("how many Philosophers? "); read(n)
  writes("how many sessions per Philosopher? ")
  read(t)
  # create the Servant and Philosophers
  var s: cap Servant
  s := create Servant(n)

  var hosts[0:2]: string[20]
  hosts[0] := "ivy"
  hosts[1] := "sleep"
  hosts[2] := "elysium"
  fa i := 1 to n →
      var vmcap: cap vm
      vmcap := create vm() on hosts[i mod (ub(hosts) − lb(hosts) + 1)]
      create Philosopher(s, i, t) on vmcap
  af

end
```

Figure 3.8: The Dining Philosopher's Main Resource

3.2 Implementation Overview

The original implementation of SR from the University of Arizona, version 2.3.1 [1], consists of three major components: a compiler, a linker, and a run-time system. In addition, this version of SR includes MultiSR, which allows SR programs to run on true shared-memory multiprocessors. The original version of SR runs on many different processor architectures and several versions of the UNIX operating system.

Documentation that describes the internals of the SR compiler and run-time system is limited to a few short articles. In Appendix E of [10], Andrews and Olsson give an overview of the original implementation, focusing on the run-time system. This discussion describes the main data structures and algorithms used to implement many of SR's interesting features such as resources, operations, and invocation handling (SR's most complicated mechanism). The overview briefly describes the thread implementation and multiprocessor scheduling.

Gebala and McNamee give a fairly detailed description of the SR compiler and run-time system in [51] (this document is included with the SR source distribution [1]). This report concentrates on the internals of the SR compiler; it describes the compiler's symbol table and parse tree. In addition, the report provides a detailed example of how the compiler transforms source code into a parse tree representation. Finally, the report gives a short discussion of the run-time system.

Finally, Townsend and Bakken provide a guide to porting SR in [111] (this document is also included with the SR source distribution). The porting guide describes some of SR's most system dependent code, such as process switching and multiprocessor support.

This section gives an overview of the original SR implementation, but does not attempt to replace the documents mentioned above. Instead, this section will elaborate on parts of the SR run-time system that have the most impact on process and synchronization performance. As such, the focus will be on threads, multiprocessor support, and network communication. The portions of the SR implementation discussed here provide a basis for later chapters.

3.2.1 Compiler and Linker

The SR compiler reads in source code and translates it to C code. It then invokes a C compiler to read the output from the SR compiler and generate machine code. The resultant machine code is referred to as generated code. By convention, the generated code interacts with the run-time system by manipulating run-time data structures and calling the appropriate run-time system primitives. The SR linker combines the generated code with the run-time system to produce an executable file. See Section 2.2 for a general description of the translation process and run-time support.

The SR compiler also performs some optimizations. For example, if a process calls a proc that is in the same resource, then under certain circumstances the compiler generates conventional procedure-call code rather than calling the run-time system (see Appendix E in [10]). A similar optimization can be performed by the run-time system at run-time to procs called in the same virtual machine as the caller. A final optimization is the conversion of operations used as semaphores to internal run-time system semaphores, which are much more efficient.

3.2.2 Run-Time System

The run-time system is fundamental to the original SR implementation. First, the run-time system provides an execution environment for the generated code. Second, the run-time system provides the implementation of all of the major language features. Finally, the run-time system interfaces to the underlying architecture and operating system. Therefore, the run-time system aids in keeping the generated code portable as it hides many system dependent details.

3.2.2.1 Structure

The SR run-time system is simply a collection of functions and data packaged into a library. However, the structure of the SR run-time system is rather peculiar when compared to other software libraries. In many ways, the run-time system resembles an operating system kernel. For instance, the SR run-time system manages many of the same resources that a typical UNIX kernel must manage, such as processes, file I/O, and network communication.

Most kernels provide an interface to kernel services through system calls. The SR run-time system provides several exported functions (primitives) that serve as "system calls". However, unlike most kernels, the run-time system resides in the same address space as the calling code. As such, both the generated code and the run-time system can operate on global data structures. In addition, the definition of the interfaces in the run-time system is much more free-form than that found in most operating systems. The run-time system primitives were not designed to be used by application programmers; they were designed to be used by compiler-generated code.

The SR run-time system is responsible for managing almost all of the language elements, including virtual machines, resources, globals, operations, invocation processing, and input/output. In addition, the run-time system provides memory allocation and timing functions. It also implements a user-level thread package so that multiple SR processes can execute within a single UNIX process or on a true shared memory multiprocessors (threads and multiprocessor support are discussed in detail below). The current version of SR is based on the UNIX operating system; therefore, the run-time system relies on many UNIX abstractions and system calls.

A description of how a program begins execution and the internals of the major language mechanisms is given in [10]. Particular attention is given to invocation processing.

3.2.2.2 Threads

SR processes are implemented as non-preemptive, user-level threads (see Section 2.4.2.1). In a sense, a significant part of the run-time system is really a user-level thread library. However, the thread system is tightly coupled to the rest of the run-time system. The thread system is built around two architecture-dependent, assembly language routines that handle the creation of a new thread and context switching from one thread to another. The two routines are:

```
sr_build_context(void (*entry)(), char *context, int size,
                 long arg1, long arg2, long arg3, long arg4)

sr_chg_context(char *newctx, char *oldctx)
```

The `sr_build_context()` function creates a new thread given several parameters. The `entry` pointer is used to designate the start function for the new thread. The start function will be passed the four arguments `arg1`–`arg4`. The `context` pointer is really a pointer to a region of memory that will be used as the new thread's stack. The size of this memory region is passed as `size`. The "bottom" of the stack is used to save the current register values on a context switch and to store a magic number, which is used to help detect memory corruption (possibly due to stack overflow).

Note that `sr_build_context()` does not activate the new thread. The new thread will only execute if it is explicitly switched to using the `sr_chg_context()` routine. When `sr_chg_context()` is called, the current register state is saved into `oldctx` (recall that the register state is stored at the bottom of the stack) and the register state found in `newctx` is restored.

These two relatively simple assembly routines are used as the foundation for SR's user-level thread system. Higher-level data structures and primitives are built using these routines. One of the most fundamental structures in the run-time system is the `proc` structure. Every active process has an associated `proc` structure. This structure is similar to a process control block (PCB) found in operating system kernels; it records process specific information that is essential to supporting SR's semantics. For example, the `proc` structure has fields for process priority, process status (e.g., ready or blocked), the owning resource, and a next pointer for linking procs onto lists. It also has a pointer to the context block (stack) that is used by the assembly routines described above. A detailed discussion of this structure and its impact on performance is given in Chapter 5.

The `proc` structure has several additional pointers that are used to put processes on different queues. One of the most important queues is the `ready queue`. This queue holds a list of processes that are ready to run. Other queues are used for blocking processes on semaphores, invocation handlers, and I/O.

All of the run-time system's process-oriented primitives are based on the `proc` struct and the assembly routines given above. Some of the most common primitives for manipulating SR processes are:

```
Proc sr_spawn (Func pc, int pri, Rinst res, Bool have_res_mutex,
               long arg1, long arg2, long arg3, long arg4)

void sr_activate (Proc pr)

void sr_reschedule (Proc pr)

void sr_scheduler (void)

void sr_kill (Proc pr, Rinst res_mutex_held)

void block(Proc queue)

void awaken(Proc queue)
```

The `Proc` type used in these functions is defined as a pointer to a `proc` struct. The `sr_spawn()` primitive creates a new SR process. The parameters required by `sr_spawn()` include a function pointer (`pc`), a priority level (`pri`), the owning resource of the process (`res`), a flag to indicate if the caller has acquired the owning resource's lock (`have_res_mutex`), and four arguments that will be passed to the new process (`arg1`–`arg4`). The `pc` and `arg1`–`arg4` are passed directly to the assembly routine `sr_build_context()`, described above. It is the responsibility of `sr_spawn()` to create a new stack for the new process. As with `sr_build_context()`, `sr_spawn()` does not cause the new process to start execution. In order for the new process to start running, two things must happen. The newly created process returned as a `Proc` pointer from `sr_spawn()` must be "activated" with `sr_activate()`. The `sr_activate()` primitive puts the given `Proc` onto the global ready queue. Even at this point, the new process does not begin executing. Only when the currently running process explicitly gives up the CPU will the new process run (if it is the next process on the ready queue). The current process can give up control by calling either `sr_scheduler()` or `sr_kill()`. The `sr_reschedule()` primitive will put the current thread onto the ready queue; `sr_scheduler()` must be called to switch to the next process on the ready queue. The `sr_kill()` primitive terminates the current process (or a specified process) and switches to the next process on the ready queue (by calling `sr_scheduler()`).

The `block()` and `awaken()` primitives help with various forms of condition synchronization. These primitives are implemented as macros because they are called frequently. The `block()` primitive puts the current process onto a specified queue. However, the current process will continue executing until it calls `sr_reschedule()`. The `awaken()` primitive will remove a process from the specified queue and put it onto the ready queue. Again, the awakened process will not begin to execute until the current process gives up the CPU and the awakened process is scheduled (using `sr_scheduler()`). The `block()` and `awaken()` primitives are primarily used to implement semaphores and invocation processing. The rest of the run-time system uses semaphores for synchronization.

Using user-level threads in the presence of blocking I/O calls is problematic. The SR run-time system uses a relatively complicated scheme to insure that a process waiting on I/O (e.g., reading a file or reading from a network stream) does not block other processes that are ready to run. The heart of this scheme relies on the UNIX `select()` system call, which allows a UNIX process to monitor several file descriptors at once. (See [103] for more information on `select()`.)

If a process performs an operation that blocks, the run-time system puts the process onto a special queue (e.g., an I/O waiting queue). Other processes are then able to run. Once there are no more ready processes (some may be blocked waiting for I/O, while others may be blocked on a semaphore or an operation), the system becomes *idle*. A special idle process is allowed to run. This process simply enters an infinite loop waiting for something to happen. Specifically, it checks to see if I/O is available for any process waiting on the I/O wait queue. Once I/O is available for a process, it is moved from the I/O wait queue to the ready queue, then it is allowed to execute. Note that the idle process also checks for program termination (system deadlock).

This method for managing I/O and user-level threads is also used to handle process sleeping with the `nap()` function (nap allows a process to go to sleep for a specified period of time) and to manage network connections with different virtual machines.

3.2.2.3 Multiprocessor Support

While the standard UNIX implementation of SR multiplexes one or more SR processes onto a single UNIX process, MultiSR allows processes to execute in parallel on shared-memory multiprocessors. MultiSR is incorporated into standard SR as conditionally compiled source code. MultiSR requires operating system support for creating kernel-level processes that execute on real processes. [2].

[2]Throughout this dissertation, we use the term "standard implementation" to denote the uniprocessor version of SR. We use this term when it is necessary to differentiate between uniprocessor SR and MultiSR.

MultiSR creates N job servers, where each job server runs on a real processor. Each job server simply tries to run SR processes (user-level threads) from the shared ready queue. All of the thread primitives and mechanisms described in the previous section remain intact. Two or more processes can execute simultaneously, possibly within the run-time system. More importantly, these processes can access and modify shared data structures at the same time. In the standard version of SR, only one process can execute at any give time, so there is no conflict with shared data. In MultiSR, all shared data must be *locked* to insure mutual exclusion to the data and to avoid interference.

MultiSR uses operating system supplied locks (usually some form of a mutex lock or a spin lock) to protect shared data. The approach taken in MultiSR is to associate a lock with individual data items. Often, the run-time system must operate on several data items at the same time, so several locks must be acquired before the data is modified. To avoid deadlock, MultiSR uses a locking hierarchy convention that specifies the order in which locks must be acquired. This ordering prevents internal run-time system deadlock.

Context switching in MultiSR is slightly more complicated than in standard SR. Using the assembly routines described above requires that on every switch to a new process, each job server must first switch to a special thread to avoid a race condition for process stacks. The result is that each logical switch requires two real context switches. A further analysis of this problem and some solutions are given in Chapter 5.

Finally, in order to preserve the complicated method of handling I/O for user-level threads, MultiSR designates one job server to be an "I/O server". That is, only one processor is allowed to perform I/O operations. If a process is running on a job server that is not the I/O server, but wants to perform an I/O operating, the run-time system puts the process onto the I/O job server queue. This can create a bottleneck for I/O operations.

3.2.2.4 Networking

Networking in the SR run-time system is accomplished with TCP/IP sockets. The networking portion of the run-time system is not invoked until an SR program attempts to create a new virtual machine. Once such an attempt is made, networking in enabled. An additional process (UNIX process) is created to run a program called `srx`, which is used to assist in the creation and destruction of virtual machines as well as to help determine when a distributed program is deadlocked. Once `srx` is running, the original (main) virtual machine sends a request to `srx` to create a new virtual machine.

To create a new virtual machine on the local machine, `srx` uses the UNIX system calls `fork()` and `exec()`. To create a new virtual machine on a remote computer, `srx` uses the UNIX remote shell command (`rsh` or `remsh`). Once a new virtual machine is created, whether it is local or remote, a new socket connection is made from the original virtual machine to the new one. That is, `srx` is only used to create a new virtual machine and establish the initial connection, after which all communication is done directly from one virtual machine to another.

Clearly, the use of these system calls and commands make the virtual machine and networking portion of the run-time system heavily UNIX dependent.

Once a connection is established, the original virtual machine may create resources on the new virtual machine. The process creating the resource is returned a resource capability, just as if the resource was created on the local virtual machine. However, invoking operations with the resource capability will cause invocations to be sent over the network by the run-time system. The remote run-time system invokes the resource just as if it was called locally. When the remote resource finishes processing the invocation, the remote run-time system returns any result back to the originating run-time system.

Internally, the remote run-time system creates a new process (thread) to handle the remote invocation. Therefore, thread creation time and context switching can have a significant impact on the performance of remote invocation. This issue is further analyzed in Chapters 4 and 5.

3.3 Summary

SR is a versatile concurrent programming language that supports multiple communication primitives. The balance between high-level and low-level primitives results in a language that can be used to solve a variety of distributed and parallel problems. This chapter has provided an introduction to SR and a description of the language implementation.

SR has served as a reference language for much of the research in this dissertation due to its unique support for multiple concurrency primitives. Chapter 4 describes several different experimental implementations of the SR run-time system. This experience motivates the analysis presented in Chapter 5. The implementation experience and analysis provides the foundation for the design principles of the Mezcla thread framework. We will see SR used again in Chapter 9 to evaluate a prototype of the Mezcla thread framework.

Chapter 4

Initial SR Implementation Experience

The initial research for this dissertation consisted of several design and implementation efforts using the SR programming language. All of these efforts included porting or re-implementing portions of the original UNIX-based SR run-time system (see Section 3.2) to new target platforms. The target platforms included the Mach microkernel [22, 23], POSIX Threads [24], the Panda parallel run-time library [94], and directly on the hardware with the Flux OS Toolkit [43]. Each of these platforms has unique characteristics, either in terms of functionality or portability, which made them useful for exploring implementation issues in concurrent programming languages.

The early investigations focused on three issues: semantic support, efficiency, and portability. In terms of semantic support and efficiency, each target was investigated from two perspectives:

- *How can a particular target benefit the SR run-time system and, consequently, SR applications?* Alternatively, how can a design of the run-time system take advantage of a specific target platform? Often the features or restrictions of a target can also have implications on the design of the run-time system.

- *To what degree do the needs of the run-time system match the functionality of the target?* This second question addresses the limitations and restrictions that make it difficult, inefficient, or impossible to implement certain aspects of the run-time system.

These two perspectives are similar, but they are subtly and importantly distinct. In the first perspective, we try to make the best out of an existing target platform. This position is common for many software developers that must build on existing interfaces (typically those provided by the operating system or function libraries). Often, this approach requires extra or awkward code to make up for the deficiencies of the existing interfaces. In contrast, the second perspective seeks to identify the source of any mismatches between the run-time system and target platform. That is, we try to identify the fundamental language requirements or target functionality that contribute significantly to a particular mismatch.

While it would be nice to have a complete custom implementation of a language and its run-time system for each desired target platform, good software engineering practices dictate that we attempt to reuse as much code as possible among implementations. Reusing code helps to minimize the complexity of large systems and also to reduce the amount of platform specific code that must be rewritten for each target, which usually increases the degree of portability of a language implementation. Therefore, another concern in the early investigations is the overall structure of concurrent run-time systems for mitigating the tension between efficiency and portability, as well as for facilitating support for diverse or non-standard targets.

A less prominent theme in the early investigations is the broader question of where a language run-time system fits in the context of a larger system. That is, what is the relationship between a language run-time system and its environment (the operating system, underlying hardware, and network)?

This chapter presents the design and implementation issues encountered with the four previously mentioned targets: the Mach microkernel (Section 4.1), POSIX Threads (Section 4.2), the Panda portability layer (Section 4.3), and the Flux OSKit (Section 4.4). The discussion of each target also addresses run-time system structure, portability, and performance.

4.1 SR on the Mach Microkernel

The initial motivation for much of the research presented in this dissertation stemmed from the apparent suitability of using a microkernel-based operating system to implement the SR run-time system. To address this question, we started designing and implementing MkSR (microkernel SR).

Microkernels have existed in many forms for several years. Mach 3.0 from Carnegie Mellon University is probably the most popular research microkernel [3]. The microkernel versus monolithic kernel debate is still an active topic in the operating systems community. For example, Liedtke presented some very impressive performance results using the L4 microkernel and argued that microkernels can be very efficient and still maintain the IPC abstraction [79]. Recently, the MkLinux project [37] exhibited the portability advantages of basing an operating system on top of a microkernel. Also, Härtig et al. ported Linux to L4 and demonstrated that complex operating system services can be implemented efficiently on top of a microkernel (the Linux port to L4 performed within 5% of native Linux) [56]. Despite these positive results for microkernels, most commodity operating systems remain monolithic.

In any case, a trend in operating systems research is to minimize the functionality of the kernel, thus allowing user-level applications to have greater control of system resources and implement higher-level abstractions. Extensible kernels [27, 41] take the microkernel concept one step further by allowing user-level applications to download code into the kernel in a safe manner.

We chose to use Mach as our experimental platform because it was the most accessible microkernel at the time we started the MkSR project. The Mach 4 kernel from the University of Utah combined with the Lites 4.4 BSD server from the Helsinki University of Technology provided a reasonable development environment for experimenting with microkernel-based software. MkLinux, a server implementation of Linux for the OSF/1 RI microkernel [37], became available well after we started developing on Mach.

The Mach microkernel [3] supports a small number of powerful abstractions including tasks, threads, ports, messages, and memory objects. A task is a passive object that encapsulates system resources. The primary resource of a task is its (virtual) address space. A task also contains threads, ports, port rights, and memory objects. Threads communicate with each other by sending messages through ports. In fact, all kernel requests are sent as messages from a thread to the kernel. The only real system calls are the ones used to send and receive messages.

In order to manipulate the Mach kernel abstractions, the programmer is given a set of interfaces and tools. The host language used to write Mach programs and servers is C or C++. However, C and C++ do not support the notion of multiple threads of execution; thus, a library package called Cthreads is required to write multi-threaded programs. Although Mach supports asynchronous message passing, the syntax and semantics associated with the system calls to send and receive messages is quite complex. Therefore, a tool called MIG (Mach Interface Generator) is used to create client and server stubs for RPC (remote procedure call) [34].

4.1.1 Design and Implementation

The Mach microkernel is extremely flexible, giving the programmer several options for implementing a server or task. It follows that there is no single method for implementing a language such as SR. The goal is to take advantage of the Mach primitives while preserving the semantics of SR. To ease the development process for MkSR, we attempted to preserve as much of the original SR implementation as possible. The following sections describe four run-time system designs for MkSR. Table 4.1 provides a summary of the run-time system designs and their fundamental differences. The first letter in each acronym specifies whether or not the run-time system uses a single Mach task or multiple tasks. The second letter specifies the use of either one Mach kernel thread or multiple kernel threads. The last letter indicates whether the run-time system maps SR operations onto "real" Mach ports or the run-time system supports operations internally, using "virtual" ports (which are really just queues).

Run-time System	Mach Tasks	Mach Kernel Threads	Mach Ports	Prototype
SSV	Single	Single	Virtual	CtUserSR
SMR	Single	Multiple	Real	N/A
SMV	Single	Multiple	Virtual	CtKernelSR
MMR	Multiple	Multiple	Real	N/A

Table 4.1: MkSR Run-time System Designs

The four approaches presented below in Sections 4.1.1.1 through 4.1.1.4 are not exhaustive, but other possible implementations will most likely be variations of the approaches given here. Additional design issues that pervade all four MkSR run-time systems are discussed in Section 4.1.1.5.

We implemented a prototype of MkSR on top of the Mach 4 microkernel. The prototype demonstrates the performance of SR processes using the SSV RTS and the SMV RTS. The prototype does not fully implement the design specifications, as inter-VM communication is not supported. To expedite the development process, we used Mach Cthreads. We present two configurations: one in which a single kernel thread is used (CtUserSR, which resembles the SMV RTS) and one with unlimited kernel threads (CtKernelSR, which resembles the SMV RTS), i.e., each SR process is mapped to a kernel thread. Table 4.1 indicates the relationship between the designs and the prototype. We compare the performance of our Mach based implementation with original SR running on top of the Lites server. The performance of the MkSR prototype is presented in Section 4.1.2.

4.1.1.1 The SSV RTS

The first, and most straightforward approach, is to implement an SR program as a single Mach task running a single Mach kernel thread using virtual ports. In the original implementation of SR, a virtual machine is associated with a single UNIX process. In a similar manner, this approach associates a virtual machine with a single Mach task (recall that a Mach task encapsulates a single address space). Because of the similarities between a UNIX process and a Mach task, this method minimizes the work necessary to port the existing SR source code to Mach. Although an SR program executes as a Mach task in the SSV design, it does not use multiple Mach kernel threads. Similarly, all message passing (operations) within a single virtual machine are handled by the original run-time system using virtual ports rather than real Mach ports. Because a Mach task represents a single address space, shared variables between resources works as expected. Similarly, the compiler and linker remain virtually unchanged. The run-time system needs only to be modified to use Mach system calls and the Mach port system to communicate with other virtual machines.

Figure 4.1 illustrates the conceptual organization of the SSV RTS. The large solid thread represents a Mach kernel thread, while the smaller dashed threads represent SR processes implemented by the original SR's user-level thread system. An SR program and run-time system are implemented using a single thread within a Mach task. The solid circles denote Mach ports, which are used by tasks to communicate with the kernel as well as other tasks. In this design, ports are only used for system calls in the run-time system and as a mechanism for virtual machines to communicate with each other. The solid boxes that enclose the resources represent a logical separation required by SR semantics: resources can only communicate with other resources through imported operations or operations that

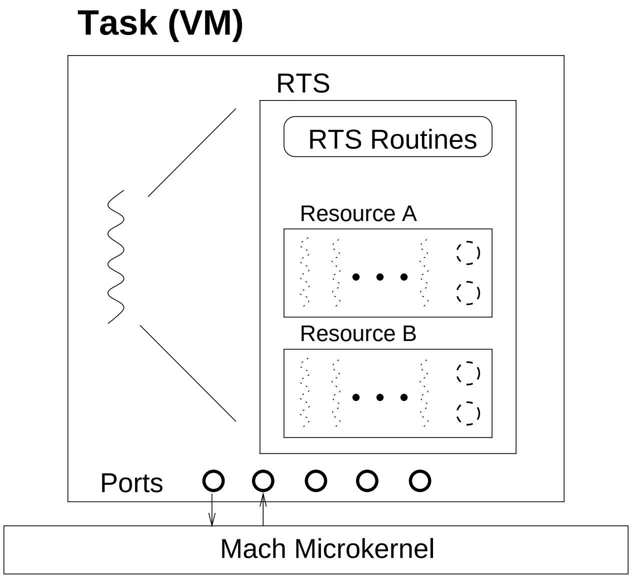

Figure 4.1: SSV RTS

are passed as arguments. Because all resources and the run-time system share the same address space, the generated code can directly access all the run-time system primitives. In addition, this shared address space approach affords a straightforward implementation of shared variables in global components. Resources that import a global component can access the global directly through a pointer, just as in the original implementation of SR.

A full implementation of SR on Mach requires two additional considerations. First, a new mechanism for creating virtual machines is needed. The original implementation uses the UNIX remote shell command (`rsh` or `remsh`) to start remote virtual machines. Communication between virtual machines is accomplished via UNIX sockets. Because an SR program is implemented as a Mach task in the SSV design, the remote shell command and sockets are not available. For communication, Mach ports can be used, but for remote execution a Mach-specific replacement for UNIX's remote shell command is needed. Second, Mach does not directly support a file system. Mach system calls are needed to interface with a file system server.

4.1.1.2 The SMR RTS

Another approach is to implement the MkSR run-time system using both Mach kernel threads and ports (i.e., real ports). Similar to the SSV approach, a virtual machine is associated with a Mach task. Unlike the SSV approach, SR processes are mapped directly to Mach threads. This approach obviates the need for many run-time system routines used to implement threads, including routines for stack allocation and management, thread scheduling, and context switching. Using Mach threads also introduces preemptive scheduling, which requires the run-time system to be reentrant. MultiSR addresses this problem by protecting internal data structures with locks. Therefore, the single task run-time system can use many of the locks employed in MultiSR. Another benefit of this approach is the potential for executing SR processes in parallel on a multiprocessor.

For message passing, SR operations can be implemented as Mach ports. In this case, rather than generating calls to the run-time system to handle message passing, the compiler will translate **send** and **receive** statements into stub procedures that will ultimately use the Mach primitives to send and receive messages on ports. SR **calls** to **procs** can be implemented in the same way with the help of MIG, which implements RPC with message passing on ports (via a send and reply port). Using Mach ports eliminates the need to implement invocation queues in the run-time system. All invocations are sent from one process to another through a Mach port.

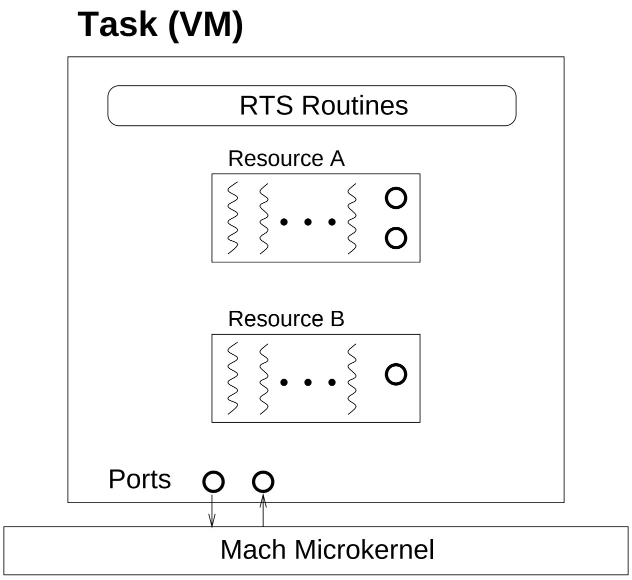

Figure 4.2: SMR RTS

Figure 4.2 illustrates the SMR RTS. An SR program is a collection of Mach threads, represented as solid lines. The solid circles located inside the resources denote Mach ports.

This organization implies that SR operations are implemented with Mach message passing. The run-time system and resources all reside in the same address space, which facilitates easy access to shared variables. Mach ports are also used to communicate with other virtual machines, rather than using a separate protocol as in the original UNIX version of SR.

Although Mach ports seem to provide a relatively easy way to implement SR operations, this approach has some fundamental problems. Consider the case where a **send** or **call** is serviced by a proc. After a **send** or **call** is executed, a new process is created to service the operation. Executing a send on a Mach port will not dynamically create a new thread, so the proc that is servicing the operation must already have a thread associated with it that is blocked (by issuing a system call to receive a message) on an operation port. This problem can be resolved by having the run-time system start up a thread for each proc in a resource when the **create** statement is executed. Each proc thread will block on a unique operation port by issuing a Mach receive message system call. When a message is sent to an operation port, the appropriate proc will be unblocked and begin execution. Unfortunately, if the same operation port is accessed again, a new proc instance will not be available to service the operation. This problem can be solved by having the original proc thread create a new thread that blocks on the same operation, which will allow multiple invocations of the same proc. When the original proc has finished executing, it will simply die. However, a large number of procs in a resource could lead to a large number of "unused" threads waiting for invocations.

Another problem with using Mach ports is the inability of threads to examine the contents of a message queue. This restriction causes difficulty in implementing the general input statement (**in**). An arm of the **in** statement can include a synchronization expression and a scheduling expression. Each requires the process servicing the operation to examine the current invocations on the queue of pending invocations to determine the next invocation to service. Consider the following example.

```
process Service
    in opa(x,y) st x > y →
        service_opa(x,y)
    [] opb(z) by z →
        service_opb(z)
    ni
end
```

The first arm of the "in" statement above selects invocations of `opa` such that the first parameter, `x`, is greater than the second parameter, `y`. The second arm selects invocations of `opb`, but it is constrained to service the invocation with the greatest value of `z`. In both cases, an invocation queue must be searched to select an appropriate invocation. While Mach IPC is asynchronous, it has no facilities for allowing user-level threads to select messages from the kernel buffers.

Without being able to examine message queues, additional run-time support is required to implement the semantics of a general **in** statement. It is feasible to have an intermediate thread that buffers invocations. Through a well defined protocol, proc instances can interact with the intermediate thread to effectively "view" the message queue. Unfortunately, this complicated mechanism may introduce a bottleneck that could inhibit parallelism. Therefore, using Mach IPC would not achieve our original goal of simplifying the run-time system.

Additionally, involving the kernel in communication that occurs in the same address space is inefficient. It is much faster to pass a pointer than to send a message with a kernel call. On a uniprocessor, at least one context switch will be required to transfer control from the sender to the receiver. Context switches in a user-level thread package are quite fast because the switch occurs in user space. However, using a Mach message requires at least two kernel boundary crossings: one to issue the send and one transfer control to the receiver.

Finally, unlike Mach, other microkernels, such as L3 and L4 [79], and Amoeba [109], provide only synchronous IPC. Even newer versions of Mach, such as Mach 4 from the University of Utah [46] and OSF/1 MK [37] have implemented synchronous, cross-address space communication to improve IPC performance. Consequently, run-time system invocation management is required by systems that provide only synchronous IPC.

4.1.1.3 The SMV RTS

The SMR RTS attempts to take full advantage of the the Mach primitives. However, as described in Section 4.1.1.2, the implementation of the generalized `in` statement presents a problem because it is not possible to examine the contents of Mach port queues. An alternative approach is to take advantage of Mach threads, but continue to use the run-time system to handle operations using virtual ports. This method is called the SMR RTS.

This hybrid approach reduces the run-time system complexity by eliminating the internal thread system, but retains the mechanism for queuing invocation blocks. This method also preserves existing code that implements the **in** statement. In some respects, this implementation is similar to the MultiSR configuration. Because Mach threads are preemptive, the SMV RTS will view threads as executing concurrently just as MultiSR executes processes concurrently. In fact, this approach can take advantage of a large portion of the code written for MultiSR to deal with reentrancy. MultiSR implements a set of locks used to protect internal run-time system data (i.e., resource instances, invocation block queues, etc.). However, unlike MultiSR, this approach does not require the concept of a job server. In MultiSR, a job server is assigned to each real processor to fetch processes that are on the ready queue.

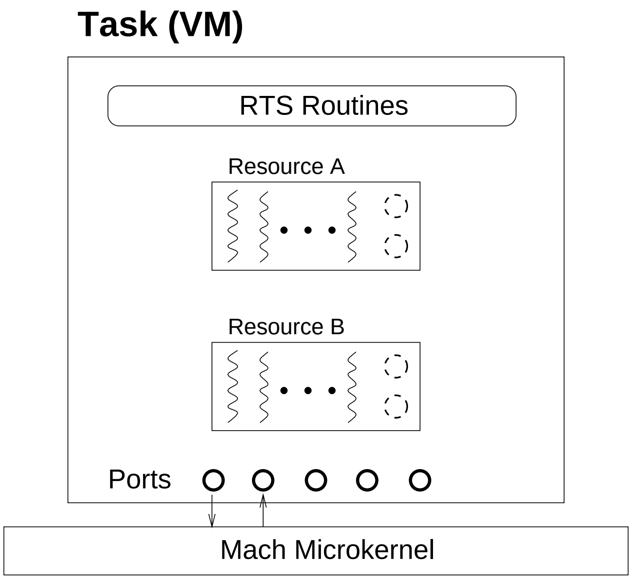

Figure 4.3: SMV RTS

Figure 4.3 illustrates the SMV RTS. Figure 4.3 is similar to Figure 4.2, except the ports (operations) inside the resources are no longer Mach ports, but rather operations managed by the run-time system using virtual ports. These ports are dashed to indicate they are not Mach ports. Similar to the SSV and SMR RTS's, all the run-time system routines reside in the same address space as the processes, making them fully accessible. All program variables are also allocated in the same address space, so pointers will operate according to the language semantics.

4.1.1.4 The MMR RTS

The single task run-time systems restrict the virtual machine to a single Mach task. Another logical organization is to associate each resource instance with a single Mach task. As in the single task run-time systems, operations can be implemented as Mach ports and procs can be handled by starting a thread for each proc in a resource instance. The run-time system proper is now implemented as a separate task. Resource processes can interact with the run-time system through dedicated ports or through shared-memory regions. Each resource task can have multiple threads representing processes. The run-time system task can also have multiple threads to service requests from the resource processes.

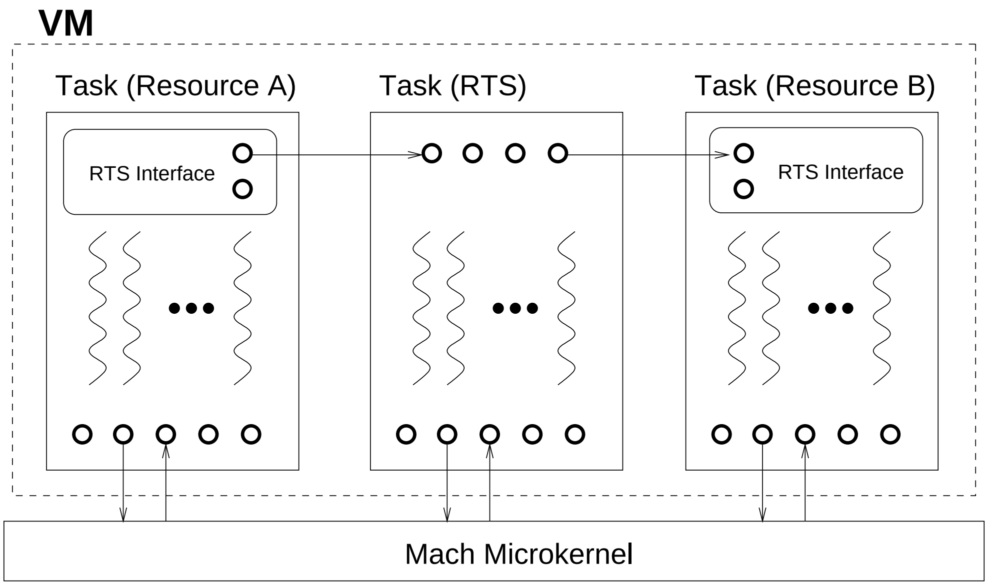

Figure 4.4: MMR RTS

Figure 4.4 illustrates the MMR RTS. Processes in different resources can communicate with other processes using operations. Depending on the type of operation, an invocation might first pass through the run-time system task. This action would occur for the generalized **in** statement, for example. For simple **in** statements, processes in different resources may communicate directly with each other through operation ports or shared memory. Also notice that a virtual machine now consists of a set of tasks.

An advantage to the MMR RTS is that local resources variables can now be accessed directly, rather than having to dereference a resource pointer. However, allowing resources to be allocated as separate tasks violates the notion of a single shared address space. (Recall that a Mach task and an SR virtual machine represent a single address space.) This fact does not prevent us from using this approach, however, because Mach allows tasks to share portions of virtual memory. Sharing can occur when a new task is created or it can occur explicitly by giving a task a port right to a range of memory. With shared virtual memory, a scheme can be developed to ensure that tasks share appropriate data and that pointers work as expected.

4.1.1.5 Additional Issues

This section addresses some additional issues that apply to all four of the previously described designs: mapping SR processes onto Mach threads, internal synchronization, and interoperability.

Mapping SR Processes to Mach Threads In the SMR, SMV, and MMR run-time systems, SR processes are mapped onto Mach kernel threads. This mapping is simple and allows Mach to schedule SR processes onto multiple processors (if available). In addition, this mapping simplifies the I/O model. If an SR process, running as a Mach kernel thread, blocks on I/O, it will not block other SR processes that can make progress.

The problem with this mapping is that Mach kernel threads are still somewhat expensive when compared to user-level threads. Both thread creation and context switch times are slower with Mach kernel threads than with user-level threads (see Section 4.1.2.1). Thread operation costs affect not only SR programs that create and switch between many processes, but also programs that perform inter-VM communication. When a virtual machine receives a remote invocation, it always creates a new process to handle the invocation; specifically, a network process receives the invocation, then starts a new process to service it.

While kernel threads can take full advantage of multiple processors and simplify I/O access, they incur a performance penalty. One solution is to use the Cthreads approach of multitasking n user-level threads on to m kernel threads where $n \geq m$. This approach minimizes the use of kernel threads and can improve the performance of programs that have more threads than available processors. This technique is similar to MultiSR's job server approach; however, in MultiSR, the number of job servers is fixed at program initiation. Cthreads allow a program to change the number of kernel threads used over time. In addition, if desired, Cthreads allows a thread to "wire" itself to a kernel thread. This functionality is useful if the thread knows it will perform a blocking I/O operation, or if it wants to ensure CPU allocation.

Multiplexing user-level threads onto kernel threads introduces complexity in both the thread system and run-time system. In the context of MkSR, the run-time system has to actively manage the number of allocated kernel threads. However, in the case of Mach, the improved performance is worth the added complexity.

The problems associated with user-level threads, kernel threads, and mapping user-level threads onto kernel threads led to a new abstraction, called the *scheduler activation*, that gives a user-level thread library tighter control over scheduling decisions made by the kernel [7] (see Section 2.4.2.3). An implementation of scheduler activations on the Mach 3.0 microkernel is presented in [21].

Finally, the overhead for thread creation and context switching in Mach is high relative to other microkernels. Systems such as L3 and L4 [78, 79], QNX [57], SPIN [27], and Exokernel [41] all exhibit superior thread performance, mitigating the negative aspects of using kernel threads.

Internal Synchronization Recall from Section 3.2 that the SR run-time system uses both simple mutual exclusion locks and condition synchronization. Mutex locks are used to guarantee exclusive access to internal data structures on multiprocessors. When an SR process is blocked waiting for a message or waiting for a rendezvous request, a form of condition synchronization is used so the run-time system can wake up the blocked process when an invocation arrives. In Cthreads, mutex variables and condition variables work well for these purposes. If the run-time system is using Mach kernel threads directly, spin locks can be used for simple mutual exclusion and IPC can be used for condition synchronization by issuing a message receive system call, which is later unblocked by a message send.

Interoperability SR programs running on top of Mach will not exist in isolation; they will need to interact with non-SR programs. System-level SR programs will require access to systems services and will also provide services. A simple way to accomplish interoperability is to map SR operations to Mach ports in such a way that the invocation and servicing of operation invocations directly uses Mach IPC.

UNIX SR provides the "external" keyword to allow SR programs to access C functions. Using the "external" keyword, additional library functions can be added that allow SR programs to send or receive messages, or perform RPC over Mach ports. By using kernel threads, calls to "external" functions that block will not affect the execution of other SR processes.

A more sophisticated approach is to provide an IDL (interface definition language) translator that can convert SR operation specifications to MIG specifications and vice versa. However, communication between SR programs and MIG based programs will be restricted to simple RPC. Finally, it is also possible to build a translator that generates C stubs directly from SR specifications. All these options require further examination.

4.1.2 Performance

As mentioned earlier, our prototype of MkSR runs on Mach 4 from the University of Utah and the Lites single server from the Helsinki University of Technology. Our hardware platform consists of an i486/66 based computer with 32MB of RAM and a 3Com 8bit network adapter.

The MkSR prototype implements SR processes and synchronization mechanisms using Cthreads. The Cthreads library allows the programmer to specify the mapping of Cthreads to Mach kernel threads. We present two configurations: one in which a single kernel thread is used (CtUserSR) and one with unlimited kernel threads (CtKernelSR), i.e., each SR process is mapped to a kernel thread. We compare the performance of our Mach based implementation against UNIX SR running on top of the Lites server.

The performance measurements in this section show the current cost of using Mach threads and the overhead introduced by the SR run-time system for operations such as thread creation, context switching, and synchronization. The measurements reflect the execution time for a particular operation. Therefore, a smaller execution time means better performance.

Many of the microbenchmarks presented in this section come from a set of benchmarks designed to evaluated the performance of SR's concurrency mechanisms [13]. The benchmarks were run on a lightly loaded system. The results are the medians of several runs, where each run consisted of 10^5 to 10^7 iterations depending on the benchmark. All of the operations in the benchmarks are parameterless. In addition, the benchmarks do not include program startup time or run-time system initialization.

Microbenchmark	CtUser	CtKernel
thread create/destroy	29.67	142.82
context switch	10.01	54.72

Table 4.2: Cthreads Performance (in microseconds)

Microbenchmark	UNIX SR	CtUserSR	CtKernelSR
local call, optimized	1.53	1.83	1.83
inter-resource call, no new process	10.60	13.50	13.90
inter-resource call, new process	40.50	180.00	767.00
process create/destroy	30.60	278.00	833.80
semaphore pair	2.31	4.08	4.08
semaphore requiring context switch	7.60	13.90	185.70

Table 4.3: Process and Synchronization Performance (in microseconds)

4.1.2.1 Processes and Synchronization

To show the relative overhead in the SR run-time system, we have measured the baseline costs of thread creation and context switching in Cthreads. In Table 4.2, the CtUser column shows times for using a single kernel thread, and the CtKernel column shows times for unlimited kernel threads.

The microbenchmarks presented in Table 4.3 measure the execution time for SR's different process and synchronization mechanisms. In turn, these measurements reveal the performance of the thread and synchronization functions provided by both Cthreads and Mach.

Both CtUserSR and CtKernelSR perform worse than UNIX SR on the same architecture. The differences in performance can be attributed to higher costs in thread creation, context switching, thread identification, and locking. Recall from Chapter 3 that UNIX SR uses a custom user-level thread package. The measurements for the single kernel thread configuration, CtUserSR, indicate that Cthreads has higher thread management overhead. This overhead reflects the greater functionality found in Cthreads. The custom UNIX SR thread system only provides architecture dependent routines for thread creation and context switching; the run-time system uses these assembly language functions to schedule SR processes. In addition, Cthreads incurs extra overhead by allowing user-level threads to be multiplexed onto kernel threads. The CtKernelSR measurements reveal the additional costs of having each SR process mapped to a kernel thread. CtKernelSR is more expensive than CtUserSR whenever thread creation or context switching is a significant part of the benchmark code.

Part of the overhead found in the Cthread implementations is due to locking. The user-level thread package employed by UNIX SR is non-preemptive: when a thread is executing, it will not be interrupted and is guaranteed exclusive access to the run-time system. Therefore, no locking of internal run-time system data structures is needed in UNIX SR. The CtUserSR and CtKernelSR implementations, however, must use Cthreads mutex locks to ensure exclusive access to run-time system data.[1]

The semaphore pair benchmark shows the extra cost of using Cthreads condition synchronization. UNIX SR's synchronization is tightly coupled with the thread scheduler, facilitating fast enqueuing and dequeuing of processes.

As described in [10], an SR local call is more expensive than a C local call because an SR call contains four hidden arguments that are used to address variables and arguments and to handle the general case that a call might be remote. Although a local call in CtUserSR and CtKernelSR requires no locking, the performance of the local call benchmark between UNIX SR and the Cthreads versions differs by approximately 0.30 microseconds. This difference is attributed to the need for thread identification. Whenever an SR proc is invoked, the process executing the proc must determine its identity in order to locate process-specific run-time system data. In UNIX SR, only one process can be running at a time, and that process cannot be preempted, so a single global pointer can be used to associate data with the current process. In CtKernelSR, however, multiple processes can be running in parallel. Each process, therefore, must be able to determine its identity in order to locate its process-specific run-time system data. Cthreads implements thread identification using the stack pointer; this method is quite efficient, although it requires that all threads have the same stack size.

The implementations measured in Table 4.3 have different levels of functionality. In particular, only CtKernelSR can take advantage of a shared memory multiprocessor. As discussed in Section 4.1.1.5, the I/O model for CtKernelSR is much simpler than for UNIX

[1]Recall that MkSR is currently designed to work with both preemptive and non-preemptive threads. Thus, even though CtUserSR does not require locking, it uses locking and thereby incurs a slight performance penalty.

Microbenchmark	UNIX SR	CtUserSR	CtKernelSR
asynchronous send/receive	16.60	22.30	22.40
message passing requiring context switch	25.60	56.80	314.00
rendezvous	54.50	109.50	318.10

Table 4.4: Intra-VM Communication Performance (in microseconds)

SR.

4.1.2.2 Communication

The microbenchmarks used in Table 4.4 measure the execution times for SR's different communication mechanisms. These benchmarks measure only intra-VM costs for virtual ports.

The performances of the three implementations differ for the same reasons discussed in Section 4.1.2.1. However, the communication benchmarks also involve SR's invocation management system, which is essential to supporting SR's communication semantics. For example, the asynchronous send/receive benchmark does not involve process creation or any context switches, so the difference between UNIX SR and the Cthreads versions is solely due to locking, condition synchronization, and thread identification.

4.2 SR on POSIX Threads

This section presents our work in developing a portable run-time system based on POSIX Threads (Pthreads) for the SR programming language. In addition to presenting specific design and implementation details of our run-time system, we also give performance results for SR implemented with three different Linux-based thread packages: a custom user-level package, MIT Pthreads [92], and LinuxThreads [74]. These results reveal general performance implications for these thread packages and the performance differences based on thread-package functionality.

4.2.1 Design and Implementation

The impetus for developing a portable run-time system grew out of two projects: MkSR (Section 4.1) and PandaSR (Section 4.3). In both projects, we started with the original SR implementation (as described in Section 3.2). It is monolithic and tightly coupled to a custom, user-level thread interface, to UNIX I/O, and to UNIX sockets. It became evident that a different run-time system structure was required to facilitate implementations for MkSR and PandaSR as well as future platforms. We sought an approach to portability that would allows us to adapt the SR run-time system to a wide range of environments (including virtual machines such as Panda [29] and microkernels). This section describes several design and implementation issues we encountered while developing our portable run-time system for SR.

4.2.1.1 A New Run-time System Structure

We based the structure for the new portable version of SR on the component approach presented in Section 2.3.4.3. The new structure is illustrated in Figure 4.5. The run-time system is now divided into system independent and system dependent components (see Figure 4.5 [2]). This structure isolates operating system dependent interfaces to threads, network communication, and I/O. (This structure does not fully realize the component approach described in Section 2.3.4.3, since we still have a separation of system-dependent and -independent components.)

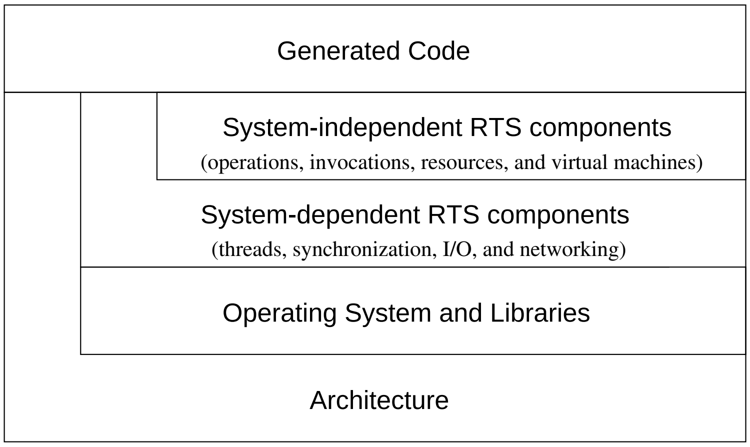

Figure 4.5: The New Run-time System Structure

[2]The virtual machines component in the system-independent level represents the SR virtual machine language construct, not virtual machines in the sense of Section 2.3.4.2

4.2.1.2 Threads

As noted earlier, an SR program may have several processes executing concurrently. In the run-time system, an SR process is implemented as a thread. However, threads can have different semantics depending on the thread package or operating system. A portable run-time system must be aware of the differences in thread systems. Examining these differences allowed us to form an abstract thread interface that can be specialized for specific thread systems. This technique results in a thread component that can be specialized without affecting dependent components. This section describes how we handle different thread characteristics and briefly discusses the issues involved in network and I/O components.

The interface to low-level thread calls is implemented as a set of macros, which are used to avoid procedure-call overhead but do not affect the basic structure of the thread component. For example, the `PROC_CREATE(pr)` macro takes a pointer to an internal SR process structure and creates a thread based on the information in the structure (i.e., function address, arguments, stack[3] , etc.). In many cases the `PROC_CREATE` macro can be simply implemented as a thread creation function call supplied by the thread system. For example, with MIT Pthreads `PROC_CREATE` is defined as follows:

```
#define PROC_CREATE(pr) do { \
  pthread_t tmp_pt; \
  pthread_create(&tmp_pt, &pt_attr_default, \
                 (start_routine) sr_intermediate_pr, (void *) pr); \
} while(0)
```

For other thread interfaces, `PROC_CREATE` may have to be more complicated, or perhaps implemented by a real function. We have also defined macro interfaces for mutex variables, condition variables, thread-specific data, thread priorities, and other thread manipulation functions.

[3]While some thread packages allocate a stack automatically at thread creation time, some do not. To handle the case in which the run-time system must allocate and deallocate the thread stack explicitly, we include a stack address field in the process structure.

4.2.1.3 Thread Scheduling

An important aspect of a thread system is whether or not it schedules threads preemptively. The presence of preemption or true multiprocessing requires the run-time system to be reentrant. That is, internal state must be protected by critical sections so that two or more threads executing in the run-time system do not interfere. In contrast, using a non-preemptive thread system requires that the run-time system selectively yield thread execution so that other threads can be scheduled.

Our portable thread component takes advantage of the MultiSR lock macros. However, we have replaced all blocking and unblocking operations previously based on the custom user-level thread package with mutex variables and condition variables. We modified the internal synchronization interfaces in this manner in order to target the most common synchronization mechanisms found in thread packages and operating systems.

4.2.1.4 Thread Creation

The way in which a thread is created varies among different thread systems. Usually, the thread creation routine takes a pointer to the function to be executed when the new thread is created and a single void pointer argument, which is passed to this function. Arbitrary data can be passed to new functions using a pointer as the argument to be passed to the function. Some thread creation routines take additional parameters including a stack pointer, the stack size, and thread attributes. Using `PROC_CREATE` (Section 4.2.1.2) hides the details of how the thread creation routine is called.

As described in Section 3.2, process creation in the SR run-time system occurs in two phases: a *setup phase* and an *activation phase*. In the setup phase an internal data structure, called a `proc`, is allocated and filled in with appropriate values (i.e., the function to start executing, arguments to be passed, a pointer to the stack, a pointer to the owning resource, and other information). After a `proc` has been setup, another routine is called to activate the proc. The activation routine calls `PROC_CREATE`, but instead of passing the actual function to create as a new thread, an *intermediate function* is passed. This intermediate function is executed before the creation of every new process and is passed as its argument a pointer to the associated `proc` data structure. The purpose of this function is to hide thread creation details from the rest of the run-time system and to aid in implementing SR's semantics by establishing a common place where internal bookkeeping (required after low-level thread creation) can execute.

The intermediate function performs two basic tasks. First, it associates the *proc* structure, passed as an argument, with the new thread. For many thread packages, this is accomplished using thread-specific data. The implementation of this association operation is hidden using the `CUR_PROC_SET` macro. Second, the intermediate function calls the actual function to be called (as designated in the `proc` structure) with four arguments (extracted from the `proc` structure). This calling convention conforms to the code generated by the SR compiler.

There are two reasons for using an intermediate function, rather than calling the function to be created directly. First, it allows the function to be called to have an arbitrary number of arguments. Many thread packages only allow one argument to be passed. Second, it performs the initial association of the `proc` structure to the thread. This association cannot take place inside the newly created function, because these functions are sometimes called directly, and not necessarily as the result of process creation.

It is possible to eliminate the intermediate function by having the SR compiler generate functions that only take a single argument, and from this argument extract other arguments as needed. Using this approach would also require the function to be called with a bit specifying whether the function was called as a result of process creation or was called directly. It is unclear if these modifications would result in more efficient code, because in the case of direct calls, multiple arguments would have to be packed in the caller and unpacked (or dereferenced via the single argument pointer) in the callee. Also, an additional conditional statement would be needed to determine if the function was called directly.

4.2.1.5 Thread Identification

Thread identification is necessary in the SR run-time system in order to associate a `proc` structure with each thread. This association is needed to allow threads to determine in which resource they are executing (so that threads can access resource variables and support other internal processing). In many thread packages, the association can be accomplished with thread-specific data.

4.2.1.6 Communication and I/O

We have only begun to develop communication and I/O components for the SR run-time system. While porting SR to Mach and Panda, we encountered different types of communication mechanisms that cannot be hidden as easily from the rest of run-time system as thread interfaces. For example, UNIX SR uses a network idle thread to poll for incoming packets. However, the Panda implementation uses an upcall to receive packets and the Mach implementation uses a blocking thread to wait for packets. Each of these networking paradigms requires specialization of the network component. In addition, Panda provides an optimized remote procedure call, which can be used to specialize the higher-level system-independent remote operation invocation code.

Our approach to I/O is to remove any notion of UNIX I/O from all run-time system components. This enables us to more easily target non-UNIX platforms, such as Windows 95, Windows NT, Mach, and Fluke [45]. A more difficult issue in dealing with portable I/O is whether or not the underlying thread package or operating system supports blocking I/O threads. If the underlying interfaces allow threads to block on I/O operations, then the run-time system is greatly simplified. Otherwise, the run-time system must incorporate extra bookkeeping to manage threads blocked on I/O. Such management requires involvement from the thread scheduler or possibly an idle thread that is similar to the network idle thread.

4.2.2 Performance

To assess the portability and performance of our new run-time system, we have built implementations using various thread packages on the Linux operating system. These thread packages include SRthreads (a custom user-level thread package based on the thread package used in UNIX SR), MIT Pthreads (version 1_60 beta6), and LinuxThreads (version 0.4). Each of these packages supports different thread features. Table 4.5 lists the prominent features of each of the tested thread packages. In the table, UNIX SR refers to the custom thread package used by the original SR implementation. The I/O support row indicates whether or not the thread system allows multiple threads to invoke I/O operations without blocking the entire program. By comparing these thread packages, we can determine the overhead of our new run-time system structure, as well as show the costs associated with using different Linux thread packages.

Feature	UNIX SR	SRthreads	MIT Pthreads	Linux
thread-level	user	user	user	kernel
scheduling	non-preemptive	non-preemptive	preemptive	preemptive
I/O support	yes	no	yes	yes

Table 4.5: Thread Package Features

We also have implementations using CThreads (Section 4.1) and Panda threads (Section 4.3), but we do not provide performance numbers here because the results are not directly comparable to the results for the Linux-based thread packages. These implementations currently run only on architectures that differ from our Linux test machine. In addition, we have ported our run-time system to other platforms such as a DEC Alpha running Linux and a DECstation 5000/240 running Ultrix.

Microbenchmark	UNIX SR	SRthreads	MIT Pthreads	Linux
local call, optimized	0.30	0.32	2.38	0.45
inter-resource call, no new process	2.35	3.01	16.51	8.68
inter-resource call, new process	10.04	12.98	115.17	532.00
process create/destroy	9.42	9.66	115.56	541.10
semaphore pair	0.48	0.50	3.37	2.41
semaphore requiring context switch	1.70	1.76	14.72	32.77

Table 4.6: Process and Synchronization Performance (in microseconds)

Table 4.6 presents the benchmark results for some uses of key language-level mechanisms (these benchmarks are the same ones used in Section 4.1.2). The test machine is a Pentium 166Mhz with 64 MBs of RAM running Linux 2.0.24. The microbenchmarks presented in this section come from a set of benchmarks designed to evaluate the performance of SR's concurrency mechanisms [13]. The benchmarks were run on a lightly loaded system and the results are the medians of several runs, where each run consisted of 10^6 to 10^8 iterations depending on the benchmark. The benchmarks do not include program startup time or run-time system initialization.

By comparing the performance results for UNIX SR and SRthreads, we see the cost of our new portability structure. Recall that SRthreads is based on the same low-level context switching code used in UNIX SR. The portable thread interface adds relatively little overhead, except for process creation. This extra overhead is primarily due to the intermediate function used to start SR processes discussed in Section 4.2.1.4.

Both MIT Pthreads and LinuxThreads perform considerably worse than UNIX SR and SRthreads. The performance difference is partially due to the added cost of thread identification and locking, which can be seen in the local call benchmark. For UNIX SR and SRthreads, the local call requires no locking and thread identification only requires a pointer dereference. However, for MIT Pthreads and LinuxThreads, the local call includes the time required to access thread-specific data. In MIT Pthreads, a global pointer (thread identifier) is used to access thread-specific data, but access to this data is protected by internal mutex variables. In LinuxThreads, the current thread's stack is used as the thread identifier, but, like MIT Pthreads, internal mutex locks are used to protect access to thread specific data.

The combined, added cost of thread identification and the locking of run-time system data structures (see Section 4.2.1.3) can be seen in the "inter-resource call, no new process" benchmark. Thread identification is needed to associate the current thread with its associated process structure on entry to a function because this information is not passed as an argument. In order to make the inter-resource call, the generated code allocates a new invocation block. The allocation pool is protected by a lock, which is required only in the preemptive thread packages: MIT Pthreads and LinuxThreads.

Comparing MIT Pthreads to LinuxThreads is interesting because both offer preemptive scheduling, but MIT Pthreads is a user-level implementation, whereas LinuxThreads takes advantage of thread support in the Linux 2.0 kernels. Locking and thread identification perform better in LinuxThreads because it uses stack-based identification and has less complicated implementations for thread-specific data and mutex locks compared to MIT Pthreads. LinuxThreads performs worse than MIT Pthreads for process creation and context switching because the kernel boundary must be crossed for these operations. It is important to note that the added process creation time and context switch time may be worth the ability to utilize true multiprocessors (something that MIT Pthreads lacks).

4.3 SR on the Panda Portability Layer

The Panda system [29] is a virtual machine (Section 2.3.4.2) for supporting parallel programming languages. Panda grew out of the Orca research group as an attempt to make the Orca distributed programming language more portable (see Section 2.2.6 for a description of Orca). In order to determine the viability of using a virtual machine for implementing concurrent programming languages, we implemented SR using the Panda portability layer. This implementation is called PandaSR.

Panda consists of two layers: the system layer and the Panda layer. The system layer interfaces with the operating system kernel primitives and is kernel dependent. The Panda layer provides a virtual machine for supporting run-time support systems. The virtual machine supports threads, message passing, remote procedure call (RPC), totally-ordered group communication, and collective communication. Panda targets run-time support systems that provide shared data or shared objects.

While virtual machines such as Panda can provide good portability, they can also hamper run-time system performance because of the use of indirection to the low-level interfaces. The fixed nature of standard interfaces or virtual machines does not easily accommodate the evolutionary and possibly radical changes in operating system architectures. Furthermore, this approach may not accommodate change at the language level.

Using a virtual machine is generally useful if one wants to take advantage of a particular communication interface. However, this approach can lead to implementations that are tightly coupled to a particular virtual machine. In addition, many languages do not require all the functionality found in such layers. Virtual machine layers can also introduce overhead in the core thread operations due to added functionality and indirection through procedure calls.

PandaSR revealed that Panda does provide good portability, but SR did not map naturally to all of the Panda interfaces. In addition, SR network communication on Panda was slower than on the native UNIX implementation.

4.3.1 Design and Implementation

For PandaSR, we replaced the custom user-level threads package with Panda threads. Incorporating Panda threads into the SR run-time system posed the same challenges faced when incorporating Mach Cthreads (Section 4.1). In fact, PandaSR was implemented at the same time as MkSR. Many of the portability changes found in the Pthreads implementation (Section 4.2) were based on earlier work from Panda and Cthreads.

SR's internal synchronization mechanisms were replaced with Panda's mutexes and condition variables. The TCP-based communication routines were replaced with message passing functions from the Panda message passing module. The `srx` process (see Section 3.2.2.4) was modified to run as a thread in the main VM (virtual machine), which is the first VM created. In Panda message passing, like many message passing interfaces, it is not possible for a node to send a message to itself. In the special case where the main VM sends a message to `srx`, a simple procedure call is invoked rather than a remote invocation. This case illustrates that the functionality of the virtual machine does not exactly match the needs of the run-time system and that extra coding is required to compensate for the mismatch.

In the original implementation of SR, VMs can be dynamically created at run time. However, in PandaSR the number of nodes used for a program must be specified at program initiation. Therefore, in PandaSR, a program can only create as many VMs as there are Panda nodes. If a VM is destroyed, the node on which it was running can be reused for another VM. Again, this mapping of VMs to Panda nodes illustrates another mismatch between the language requirements and the virtual machine. This mismatch has some significant consequences. First, the PandaSR implementation will not be able to execute all programs intended for the original SR implementation if the number of nodes created at program initiation is less than the maximum number of dynamic VMs allowed by the original run-time system. Second, unlike the original implementation, multiple VMs cannot be created on the same node in PandaSR. Finally, since Panda nodes must be preallocated, memory is wasted on unused nodes at run time.

When an SR program invokes an operation, the run-time system creates an "invocation block," which is simply a region of memory. If the operation is serviced on a remote VM, the invocation block is transferred via the "write" system call through a TCP socket. The Panda message passing module, however, operates on Panda messages. In order to avoid extra copying in PandaSR, all invocation blocks are created as Panda messages. This change is not problematic, but it does incur a small penalty for local invocations.

4.3.2 Performance

To determine performance of PandaSR, we measured the execution time of some microbenchmarks on two 70 MHz SPARCstation 5 workstations with standard 10Mbit/sec Ethernet adaptors, running Solaris (see Table 4.7). Each microbenchmark represents a different form of interaction between two virtual machines. The figures given for the original implementation come from MultiSR, the multiprocessor version of SR. MultiSR was used because it requires locking of internal data structures, which is also done in the Panda implementation[4]. All of the microbenchmarks measure the execution time of "null" operations. That is, the operations are invoked without parameters and do not have return values.

Benchmark	Original SR	PandaSR
RPC, new process	4.53	6.68
rendezvous	4.71	7.74
asynchronous message passing	1.92	3.77

Table 4.7: PandaSR Communication Performance (in milliseconds)

The RPC benchmark measures the time it takes to invoke a null remote operation. On the remote virtual machine a new process is created to service the operation, which replies on completion. The rendezvous benchmark consists of a process invoking a remote operation that is serviced by waiting processes on the remote virtual machine. The asynchronous message passing benchmark measures the time required to send a null message from one virtual machine to another.

Operation	Original SR	PandaSR
process create/destory	160	688
context switch	63	119

Table 4.8: Panda Thread Performance (in microseconds)

The performance differences between the original implementation and PandaSR are mostly due to thread creation and context switch times (see Table 4.8). The original system uses a custom user-level threads packages, whereas the PandaSR uses Solaris threads. For example, process creation in the original implementation is 160 μs versus 688 μs in the PandaSR. Also, a context switch in the original implementation is 63 μs and 119 μs in the Panda implementation. Additional overhead is attributed to the procedure call overhead to the Panda synchronization functions and the use of thread specific data to access internal

[4]In retrospect, the uniprocessor implementation of SR should have been used for comparison rather than MultiSR. This would have resulted in faster times for the original implementation.

information associated with each SR process (see Chapter 5 for detailed description of this problem).

4.4 SR on the Flux OS Toolkit

Unlike the previous target platforms described in this chapter, the Flux OS Toolkit (OSKit) [43] from the University of Utah was used to develop an implementation of SR than executes on the bare hardware without any operating system support; we call this implementation SR/OS. The OSKit is a set of libraries and tools for building operating system kernels. The toolkit provides low-level boot code, processor initialization, a minimal C library, and framework that allows device drivers to be imported from other kernels (e.g., Linux and FreeBSD). More recent versions of the OSKit include support for threads, file systems, and protocol stacks.

Implementing SR directly on the hardware enables two lines of research: determining the suitability of a communication-oriented programming language for OS implementation and designing the run-time system to support such a language directly on the hardware. Suitability can be defined by performance and usability. A crucial question to be answered is whether or not an SR program combined with the SR run-time system can effectively and efficiently manage a machine's resources. While SR's concurrency semantics have proven to be flexible and efficient for applications [13], there is no study of the performance of SR's mechanisms for implementing kernel services.

SR has mostly been used for application programs, simulations, and teaching concurrency. One significant effort that utilized SR for system software was the Saguaro distributed operating system [12]. A UNIX-based implementation of SR was used to implement a prototype of the Saguaro distributed file system. During the Saguaro project, a direct hardware implementation of SR was considered, but it was never pursued due to the lack of good development tools [9]. While the UNIX-based prototype allowed SR's mechanisms to be evaluated from a usability standpoint, it did not allow true performance viability to be assessed. A very early version of SR was implemented directly on a PDP-11/23 and several LSI-11 TERAKs, but its development and debugging were extremely tedious.

Implementing SR directly on the hardware allowed us to eliminate operating system overhead and to examine the issues of running SR applications without high-level services. We built a prototype relatively quickly and acquired some very basic performance measurements for network communication.

4.4.1 Design and Implementation

For the prototype version of SR/OS, we focused on optimizing network communication performance given the removal of any kernel abstractions. SR's mechanisms for communication and synchronization are rather sophisticated when compared to the mechanisms for such operations found in many operating system kernels. For example, as discussed in Chapter 3, SR has a fairly complicated run-time system to help map SR's semantics onto UNIX interfaces (i.e., sockets, `select()`, and non-blocking I/O).

Because the original implementation of SR is tightly coupled to UNIX I/O and UNIX sockets, we started with the more platform neutral implementation described in Section 4.2. That work removed many UNIX dependencies and isolated system-dependent functionality such as threads, synchronization, and network communication. Using this base, we extracted the thread system from the run-time system and made it an independent library, called *SR Threads*. This thread system is integrated with the OSKit's timer facility to provide a system clock and to wake-up sleeping threads.

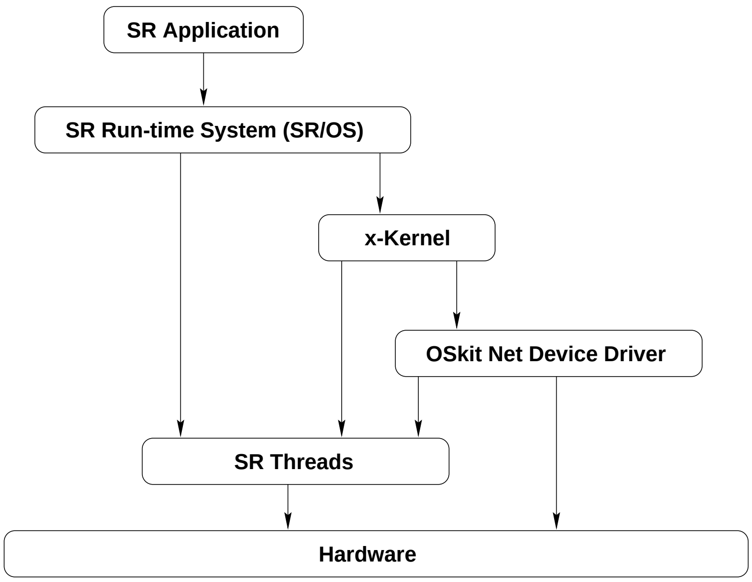

Figure 4.6: SR/OS Organization

At the time we were developing SR/OS, the OSKit did not have network protocol support. Therefore, we ported the x-kernel [62] to the OSKit. Both the SR run-time system and the x-kernel rely on SR Threads for scheduling. Currently, OSKit only directly uses functions from the Minimal C Library and from the device driver component (networking only). These components depend on other OSKit components (e.g., libkern for kernel mode initialization and lmm for simple memory management) and consequently are linked with the SR run-time system to produce the final kernel image. The organization of SR/OS is illustrated in Figure 4.6.

Because the x-kernel uses a process-per-message model, using the same thread system for both the SR run-time system and the x-kernel allows a single thread to "carry" a message all the way into the run-time system without any context switches. This type of optimization is not possible when the run-time system is built on top of an operating system kernel.

Implementing SR/OS with the OSKit was fairly straightforward. The OSKit's POSIX environment allowed most of the SR run-time system to be built unmodified, while the OSKit's default initialization and management of page tables and interrupts greatly simplified the task of running SR directly on the hardware. Once a uniprocessor version of SR/OS was complete, adding network support with the x-kernel was also fairly easy, with most of the development time spent on interfacing with the x-kernel. The OSKit allowed us to tightly couple the SR run-time system with the x-kernel using a common thread interface.

4.4.2 Performance

We measured the performance of two of SR's invocation mechanisms, remote procedure call (RPC) and rendezvous (see Table 4.4.2). All the experiments were run on two 200MHz Pentium Pro machines connected by a 100Mbps ethernet link. The UNIX SR tests were run on FreeBSD 2.1.5. Times are in microseconds. Similar to the PandaSR measurements (see Section 4.3), both operations are "null" operations. That is, they were invoked with no arguments or return values. However, a null RPC in SR/OS requires 104 bytes, which comprise internal protocol information.

In comparison to a base time on Unix/TCP of approximately 430 microseconds, SR/OS was 30% faster with TCP, 45% faster with udp, and 48% faster with IP. The SR/OS() tcp time reflects an expected improvement by removing the user/kernel boundary and extra OS code on the critical path for sending a network packet. However, UNIX SR and SR/OS use two different protocol stacks: UNIX SR uses the FreeBSD protocol stack

Benchmark	UNIX SR (tcp)	SR/OS (tcp)	SR/OS (udp)	SR/OS (ip)
Null RPC	426	302	236	220
Null Rendezvous	438	315	247	231

Table 4.9: SR/OS Round-trip Operation Invocation Latencies (in microseconds)

(which is based on the 4.4 BSD kernel [73]) and SR/OS uses the x-kernel. Research suggests that the performance difference between the two protocol stacks is not drastic [62], thus it is reasonable to assume most of the difference comes from OS kernel overhead. Further experimentation is needed to precisely determine the sources of the overhead (e.g., context-switching, system call processing, device driver performance, etc.).

4.5 Summary

The implementation experience presented in this chapter revealed many important issues for concurrent run-time systems and for the various target platforms. In particular, SR's message passing semantics makes it difficult to directly map SR's communication mechanisms directly onto target message passing systems such as Mach IPC and Panda message passing. This mismatch is a result of SR's rich semantics and the rather fixed nature of the target platforms.

With the exception of the OSKit implementation, all the other implementations resulted in poor thread performance when compared to the original uniprocessor implementation of SR. The SR run-time system uses threads extensively, thus thread performance is often critical to the performance of SR applications. The following chapter takes a closer look at the problems with current approaches to implementing threads in concurrent run-time systems.

Chapter 5

Threads in Run-Time Systems

Chapter 4 explored the intricacies of porting the SR run-time system to a variety of platforms. A recurring issue involved how to map the SR process abstraction to a particular thread implementation. Because the process abstraction is an integral part of the SR language and threads are an integral part of the run-time system, they can have a significant impact on the performance of many SR programs. In most cases, using standard thread systems resulted in poor thread performance. Our results, which are consistent with those of other researchers, show that custom, user-level thread systems almost always outperform standard thread systems [54, 7, 82, 91, 35].

The SR implementation experience revealed some fundamental problems with current approaches to implementing threads in concurrent run-time systems. The main observations are that different languages have very specific thread requirements and that different platforms [1] have varying degrees of functionality for supporting thread systems. By trying to map SR processes onto Pthreads (see Section 4.2), we learned:

- Extra and somewhat awkward code was needed to adapt the Pthreads interfaces to the thread requirements of the SR run-time system.
- The Pthreads library supports a large amount of functionality unneeded by SR.

Not only is there a mismatch between SR processes and Pthreads that leads to extra glue

[1] Recall from Chapter1 that the term *platform* is used to denote a hardware and software combination that makes a complete computer system, including the processor architecture, bus architecture, memory system, operating system, supporting libraries, and possibly the network interconnect. In the case of a bare machine, a platform refers to just the hardware configuration and possibly some minimal support software.

code, but the Pthreads interface provides more than is needed, which can also result in performance degradation.

This chapter makes the case for building custom user-level thread systems that directly map the thread requirements of a particular run-time system to the functionality of a particular platform. We leverage off the implementation experience presented in Chapter 4 by categorizing and evaluating the different techniques used to incorporate threads into concurrent run-time systems. To support the analysis presented in this chapter and to ensure that the analysis is applicable to other languages, we also refer to the implementation of Kaffe [116], a free Java virtual machine. We identify some of the key problems when implementing threads in both SR and Kaffe, including the issues of thread control blocks, context switching, and thread scheduling.

5.1 Thread Implementation Alternatives

As mentioned in Section 2.2, many imperative languages, such as SR, Java, and Modula-3, provide constructs for concurrency. Implementations of these languages use various techniques for mapping concurrency to threads. Two common techniques are: *custom threads* and *Pthreads-style threads.*

The *custom threads* approach usually produces the most efficient implementations because they are implemented as user-level threads. As we found in Chapter 4, user-level threads almost always outperform kernel-level threads and custom threads are tuned for a particular run-time system. Both SR and Java have implementations that use custom threads [116, 107].

Packages exist to help construct custom threads systems [70, 85]. A fairly portable technique for implementing user-level threads is to use the C Library routines `setjmp()` and `longjmp()`. Run-time systems based on these low-level thread building blocks generally perform well, but support only a few target configurations.

As described in Section 2.4.2.1, two notable problems with user-level threads are the lack of blocking I/O support and the mismatch between the kernel scheduler and the user-level scheduler. The former problem requires sophisticated support to allow user-level threads to block on I/O operations. The latter problem can lead to poor performance on shared-memory multiprocessors if the kernel preempts a user-level thread that is holding a lock needed by other threads. Techniques such as Scheduler Activations [7], First Class

User-Level Threads [82], and Strands [101] have been used to enable better coordination between user-level scheduling and kernel scheduling (see Section 2.4.2.3). Unfortunately, all these techniques require kernel support and are currently restricted to research systems or private kernel interfaces.

The *Pthreads-style* approach is attractive from a portability standpoint (see Section 2.4). Most operating system vendors supply a Pthreads library, which supports kernel-level threads or multiplex user-level threads onto kernel-level threads [91]. Also, many other thread systems and parallel run-time systems provide a Pthreads-style interface, such as Mach Cthreads [35], Georgia Tech Cthreads [96], Nexus [49], and Panda [29]. However, our experience has shown that Pthreads libraries generally perform much worse than custom thread implementations for basic thread operations such as thread creation, context switching, and blocking. Even Pthreads-style libraries implemented entirely in user-space tend to perform much worse than custom user-level threads (see Sections 4.1, 4.2, and 4.3). Poor performance in Pthreads-style libraries is partly attributed to the generality in the Pthreads-style interface. In addition, using a Pthreads-style interface usually means relinquishing scheduling control. Such control may not be necessary for applications that use Pthreads directly, but a language's run-time system can often make better scheduling decisions because it knows the relationships between threads, especially threads used internally by the run-time system. Also, on a multiprocessor, it is possible for the run-time system to schedule threads that use the same code and data on the same processor to exploit cache affinity [113].

Parallel run-time libraries usually provide some type of thread interface. Nexus [49] and Panda [29] have Pthreads-style interfaces; Converse [68] has a slightly lower-level interface built on top of QuickThreads [70]. Using a parallel run-time library is generally useful if one wants to take advantage of a particular communication interface. However, this approach can lead to implementations that are tightly coupled to a particular parallel run-time system. In addition, many languages do not require all the functionality found in such run-time systems. Parallel run-time systems can also introduce overhead in the core thread operations due to added functionality and indirection through procedure calls.

The approaches above have many variants. In particular, several thread packages support varying degrees of customization [26, 68, 54]. Section 2.4.3.1 described OpenThreads [54], a user-level thread package that provides Pthreads-style interfaces and a meta-interface that can be used to modify the behavior of the basic thread abstractions. Among other things, the meta-interface allows a client to modify the stack allocation function, to customize the queueing mechanism for thread scheduling, and to add callback functions to various thread events for tracing and debugging. OpenThreads is a good example of a thread package that allows a reasonable degree of customization. However, most customization is done through indirection (callback functions), the main thread interfaces are still Pthreads-style, the implementation has internal state not directly visible to the client, and there is no direct support for multiprocessors.

5.2 Experience with Threads in the SR Run-time System

As described in Section 3.2, the SR run-time system uses threads not only to implement concurrency at the language-level (SR *processes*), but also to implement internal functionality. For example, threads are created to handle remote invocations and an idle thread is used to monitor I/O, to handle timer events, and to detect deadlock. Therefore, good thread performance is essential to the performance of many SR programs.

Chapter 4 described our experience with porting and re-implementing portions of the SR run-time system to support a wide range of platforms. Section 4.2 presented a portable SR run-time system that uses Pthreads-style interfaces and a more abstract interface to remote communication.

By using a Pthreads-style interface, we gained portability, but we also introduced overhead, especially with respect to thread creation and context switching. Table 5.1 summarizes the benchmark results for some uses of key language-level mechanisms. The experiments were executed on a Pentium 166MHz machine running Linux 2.0.24.

Microbenchmark	Original SR	MIT Pthreads	LinuxThreads
process create/destroy	9.66	115.56	541.10
semaphore requiring context switch	1.76	14.72	32.77

Table 5.1: SR processes implemented with Pthreads (in microseconds)

The experimental results show the performance of the mechanisms in the original implementation of SR (which has a custom user-level thread system) and two Pthread libraries: MIT Pthreads [92] (a user-level implementation) and LinuxThreads [74] (a kernel-level implementation). These microbenchmarks are based on a set of benchmarks designed to evaluate the performance of SR's concurrency mechanisms [13].

In addition, we had to give up scheduling control as well as some aspects of SR's semantics due to the inability to support thread cancellation. The problem arises because SR allows a programmer to **destroy** a resource or a virtual machine. Destroying a resource not only reclaims the memory associated with a resource, but also terminates any processes associated with the resource. Destroying a virtual machine causes all of the resources instances on the virtual machine to be destroyed. According to the original language specification, destroying a resource requires the ability to asynchronously terminate processes (threads). Many Pthread-style packages do not support asynchronous thread termination[2].

Since Pthreads-style packages do not expose the global state of threads, we had to implement extra bookkeeping to keep track of ready threads, blocked threads, and internal run-time system threads. We also ran into problems with SR's idle thread. In the original implementation, the idle thread runs when no other threads are ready to run. As noted earlier, the idle thread is used to help detect if the running program is in local and global deadlock (distributed). We were using Solaris threads, and priorities did not work as expected. Therefore, we ran the idle thread periodically, which introduced two problems. First, the idle thread ran needlessly when there were ready threads. Second, to reduce the cost of running the idle thread too frequently, we used a longer period. However, the longer the period, the longer it takes to detect global deadlock.

5.3 Issues for Threads and Run-time Systems

Based on our experience with SR and an evaluation of Kaffe, we have identified several implementation issues that can affect both performance and functionality. Below, we outline the key issues.

5.3.1 Thread Control Blocks

The run-time systems for both SR and Kaffe have the notion of a *thread control block* (TCB). The TCB encapsulates language specific information about a thread as well as information for scheduling and blocking. For example, SR processes (threads) execute in the context of an SR resource (a dynamic module). Therefore, the SR TCB encodes the

[2]Most operating systems vendors now support Pthread cancellation. However, most Pthreads-style libraries still do not support general thread cancellation.

owning resource in addition to other language-specific fields (see Figure 5.1 for a partial listing of the SR TCB). Several fields in the TCB are also dedicated to supporting thread scheduling, such as a priority field, queue pointers, and a stack pointer. The Kaffe virtual machine [116] has a similar TCB layout.

```
struct tcb_st {                 /* process control block */

    enum pr_type ptype;          /* type of process */
    enum in_type itype;          /* type of invocation if a PROC */
    char *pname;                 /* proc name, or "body" or "final" */
    char *locn;                  /* current source location */
    int priority;                /* process priority */
    Sem wait;                    /* sem for initial completion */

    Ptr   stack;                 /* process context and stack */
    Mutex stack_mutex;           /* is the stack free yet? */

    int status;                  /* process status */
    Procq *blocked_on;           /* list process is blocked on */
    Bool should_die;             /* is someone trying to kill this? */
    Sem waiting_killer;          /* if so, is blocked on this */

    Rinst res;                   /* resource process belongs to */
    ...
}
```

Figure 5.1: Partial Thread Control Block for SR

Two observations can be made from examining TCB usage in run-time systems. First, run-time systems that employ custom user-level threads encode scheduling information inside the TCB. Adding new or different scheduling functionality generally will require modification of the TCB. For example, MultiSR, a multiprocessor version of SR [111], adds a stack lock to the TCB. Second, if a Pthreads-style thread package is used, then some thread state is hidden and a run-time system has to maintain its own thread state. Therefore, two different TCBs are required: one for the run-time system and one for the thread package. For example, because the SR run-time system needs to keep track of blocked threads, it uses two queues: one for the run-time system and one for the thread package. Not only are queues duplicated, but both queues will require separate locks when running on a multiprocessor.

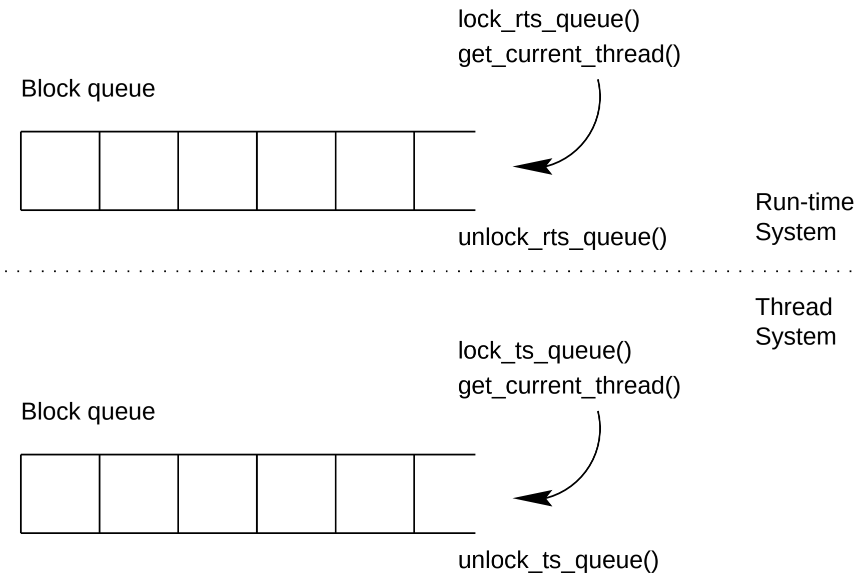

Figure 5.2: Duplication in Thread Control Block Queues

Figure 5.2 illustrates this problem. Because a language run-time system does not have access to the thread state of a Pthreads-style library, the run-time system must duplicate some of the thread system's functionality. In this case, the run-time system is keeping track of where a thread is blocked. `get_current_thread()` returns a handle that is used to relate the current thread to the run-time system TCB. The TCB is then queued. On a multiprocessor, this operation must be protected by locks. The same operation is also carried out in the thread system.

Similarly, in Kaffe, thread groups are used to keep track of collections of threads for combined operations. The two TCBs will also have to be related, so some form of thread-specific data is required to associate the run-time system's TCB with its corresponding thread in the thread package. This creates a level of indirection to the language-specific TCB information.

5.3.2 Context Switching

As described in Section 5.2, basic thread performance can have a significant impact on the performance of programs written in concurrent languages. Many thread packages and run-time systems use the same context switch code for different types of targets (e.g., uniprocessors and multiprocessors). While this may lead to better portability, the resulting run-time systems for certain targets will exhibit less than optimal performance.

The original SR implementation uses a *direct* context switch to pass control from one thread to another. The direct switch simply saves the register state and stack pointer of the current thread in a context array located at the beginning of the current thread's stack. The new thread's register state and stack pointer are restored from its context array. However, in MultiSR, when a thread blocks, two context switches are required to prevent a race condition involving the blocking thread's stack. An intermediate thread with its own stack is used to unlock the blocking thread's stack and to switch to a new thread. Thus, every thread switch requires two context switches.

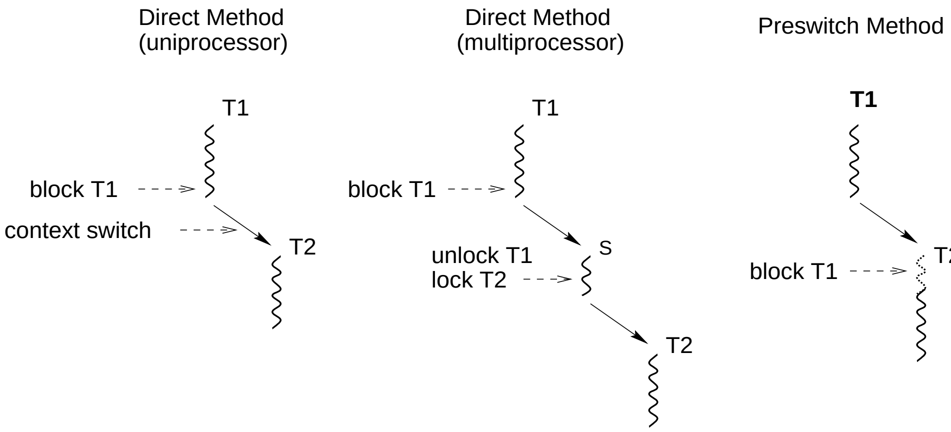

Figure 5.3: Context Switching Methods

Figure 5.3 illustrates the different types of context switching. The direct method saves T1's state and then restores T2's state. The preswitch method saves T1's state, switches to T2's stack, and then runs a function on behalf of T1. After the function returns, T2's state is restored.

Context Switch Type	133MHz Pentium	50MHz Sparc 20
uniprocessor	2.18	2.80
multiprocessor	13.74	19.28

Table 5.2: SR Context Switching Costs (in microseconds)

A simple SR microbenchmark illustrates this cost (see Table 5.2). On a computer with a 133MHz Pentium processor, a thread switch caused by blocking on a semaphore takes 2.18 microseconds with uniprocessor switching and 2.80 microseconds with multiprocessor switching, a 28% increase. The same benchmark on a 50MHz Sparc 20 requires 13.74 microseconds and 19.28 microseconds, a 40% increase. For these benchmarks, we turned off MultiSR's low-level locks in order to better isolate the performance of context switching methods and to provide a fair comparison between the Pentium and Sparc implementations. The current implementation of MultiSR for the Sparc uses relatively heavy-weight LWP (light-weight process) mutex locks. With the heavy-weight locks enabled, their cost dominates the cost of context switching. Less expensive locks could be used in MultiSR for the Sparc; they just have not been implemented yet.

By using a *preswitch* approach, in which code for the blocking thread is executed on the stack of the new thread (as used in QuickThreads [70]), the second context switch for the multiprocessor thread switch can be eliminated. Keppel also describes several other techniques for efficient context switching in different settings in [70]. Additional considerations include preemptive versus non-preemptive scheduling and lock holding across context switches. The important point is that different targets require different context switch code for optimal performance, but most thread packages do not offer a choice.

5.3.3 Scheduling and Synchronization

Researchers have long recognized the need for application specific synchronization and scheduling [26]. Likewise, language run-time systems have special synchronization needs. For example, the SR run-time system does not need direct support for condition variables. Instead, blocking is achieved with low-level queue operations and semaphores based on these operations. Rather than introducing a level of indirection by forcing the SR run-time system to use condition variables, it makes more sense to support SR's synchronization as directly as possible based on target primitives. For example, a `fetch-and-add` instruction can be used to implement fast semaphores.

Language run-time systems can also benefit from custom scheduling. For example, a run-time system can directly schedule threads to handle incoming network requests to reduce latency. Also, a language run-time system can schedule threads based on run-time knowledge to take advantage of processor affinity on a multiprocessor.

5.4 Summary

This chapter generalizes some of the results found in Chapter 4. The most significant problem is that concurrent run-time systems do not map well to platform-supplied thread systems. In addition, we identify some fundamental problems with the standard use of Pthreads-style libraries and the common way of implementing custom user-level threads. Our analysis indicates that we need a way to easily generate *multiple* custom, user-level thread systems that are specific to a particular language run-time system.

This observation motivates the design of the Mezcla thread framework presented in the next chapter. The framework is an attempt to alleviate the problems presented in this chapter by allowing a language implementor to easily create specialized thread systems that map a specific run-time system to a specific target. The goal of the framework is to automatically generate custom thread systems from a single specification file that perform as well as, or very close to, handcrafted thread implementations.

Chapter 6

The Mezcla Thread Framework: Design Issues

To address the thread implementation issues discussed in Chapter 5, we have designed the *Mezcla thread framework*. This framework generates specialized thread systems that tightly integrate language or library-specific thread data and operations with platform-specific thread functionality. We call the language run-time system or high-level thread library the *client*, and the platform and its thread functionality the *target*. The goal of the framework is to separate client-specific thread concerns (e.g., language semantics, synchronization, and scheduling) from target-specific concerns (e.g., context setup, context-switching, multiprocessor locking, and multiprocessor scheduling). A thread generation tool combines client-specific thread requirements with a specific target platform to create a specialized thread system with custom primitives to be utilized by the client run-time system.

The framework supports the generation of custom scheduling and synchronization primitives. However, Mezcla does not introduce any fundamentally new mechanisms. Instead, it uses an intermediate form and target specifications to take advantage of proven techniques that work well for different thread operations on different targets. Figure 6.1 illustrates the overall organization of the Mezcla thread framework.

This chapter examines key design issues for the Mezcla thread framework. First, it presents a design overview of the goals, assumptions, and challenges of the framework. Second, it introduces the basic concepts of the framework, including the *Mezcla abstract*

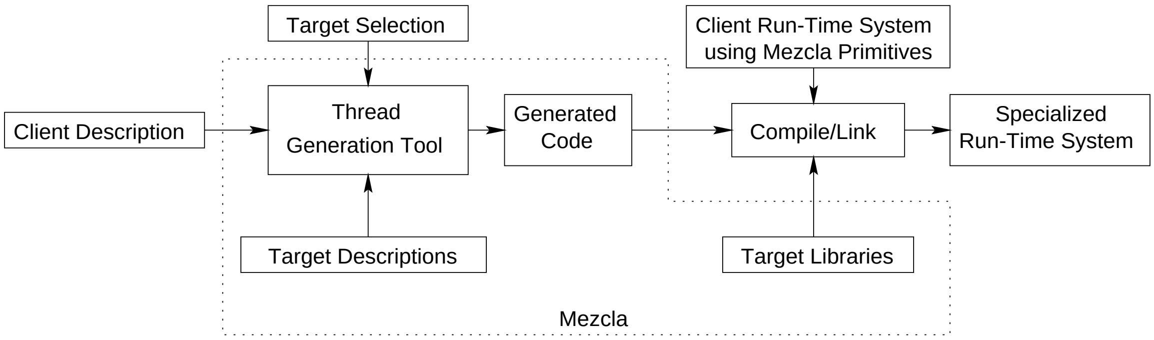

Figure 6.1: The Mezcla Thread Framework

thread model and the *Mezcla specialization model.* Third, it presents design issues for the Mezcla thread primitives. Fourth, it discusses several target issues. Fifth, it outlines issues for thread generation, including the client description language, the target description language, and implementation alternatives. Finally, it examines frameworks found in other domains that have influenced the design of Mezcla.

6.1 Design Overview

6.1.1 Goals

The implementation experience with the SR run-time system presented in Chapter 4 and the analysis of threads presented in Chapter 5 revealed many deficiencies with current techniques for incorporating threads into run-time systems. The Mezcla thread framework attempts to overcome these deficiencies. The most fundamental design goals for Mezcla are:

- **Specialized Thread Systems** The primary goal of the Mezcla thread framework is to facilitate the creation of language-specific thread systems optimized for a specified target platform. These thread systems should perform as well as (or very close to) a custom thread system for the desired language/target pair. Specialized thread systems will be automatically generated from a single specification file. This approach avoids the mapping and performance problems that result from using standard thread libraries and the portability deficiencies found in custom approaches. The remaining goals are all related to this primary goal.

- **Minimal Critical Paths** A Mezcla specialized thread system will provide a set of fixed and custom thread primitives that are tailored to a particular language and target platform. Therefore, a goal of the Mezcla thread generation tool is to produce primitives with minimal critical paths. We want Mezcla generated threads to be specialized without using indirection (e.g., callbacks).

- **Portability — A Client Description Language** The implementor of a given language L will provide Mezcla with a client description file, written in the Mezcla client description language, that details L-specific thread characteristics. This client description language needs to be both simple and expressive. The client description language serves as the portability mechanism for L's run-time system.

- **Retargetability — A Target Description Language** The Mezcla thread framework requires a uniform approach for codifying the functionality of a target platform. The Mezcla thread generation tool will combine a specific client description with a specific target description, written in the Mezcla target description language, to produce a specialized thread system. Target descriptions will be provided by an implementation of the Mezcla thread framework.

6.1.2 Assumptions

While the Mezcla thread framework is intended to be a general tool, suited to a wide range of languages and target platforms, we do make several assumptions for the design presented in this chapter. Making these assumption explicit allows us to focus the design and to better understand the limitations of the framework. The major assumptions for the design of the Mezcla framework are:

- **Language Domain** We focus on *imperative languages with semantic support for concurrency*, such as the languages described in Section 2.2 and the SR programming language as described in Chapter 3. These languages often have non-trivial thread semantics that do not map directly to standard thread interfaces; thus, these languages are good candidates for using specialized threads. We are not directly considering other language paradigms, such as functional or logic, and implicitly parallel languages. However, we expect the Mezcla thread framework will eventually be useful for any language that requires a thread system.

- **Library Domain** Mezcla was originally intended for language run-time systems. However, as will be demonstrated in Section 9.1, Mezcla can also be used to create specialized thread libraries. In this case, the Mezcla generated primitives can be used by a thread library to implement higher-level functionality or they can be used directly by an application program.

- **Target Domain** Mezcla is geared toward platforms with commodity operating systems and commodity processors as well as shared-memory multiprocessors.

- **Specialization Approach** The Mezcla approach to specialization is a purely *compile-time* approach. The goal is to generate a static library that is suited for a particular language and target. Any dynamic optimization (such as stack and TCB caching) must be handled by the client or the target description. Our view is that a language implementor will build multiple run-time systems, each customized for a different target. The application program will pick the most appropriate run-time system at program load time or link time.

- **Scheduling** The Mezcla framework is designed to support most non real-time scheduling policies (e.g., simple round-robin, priority, and any custom policy that picks a thread based on the current state of the system). Real-time thread scheduling requires sophisticated target support not found in most commodity operating systems.

- **Thread Targets** The Mezcla thread generation tool is designed to produce custom, user-level threads. As described in Chapter 8, the Mezcla prototype can also generate thread systems that are based on Pthreads. A Mezcla thread system based on Pthreads ignores most of the client specific scheduling information. Because the Pthreads target assumes the Pthreads library will control thread scheduling, the resulting thread system may not function exactly as specified by the client.

6.1.3 Challenges

The goals and assumptions listed above roughly define a design space that will help guide the development of the Mezcla thread framework. Given this design space, the difficult challenges for the framework are:

- **Separating Concerns** The framework must separate language-specific thread concerns from platform-specific concerns. When a language implementor develops a custom thread system, either there is no separation or a run-time specific, machine-independent interface is defined. In the former case, the implementor couples language-specific thread semantics with a particular platform target. In the latter case, the implementor defines an interface to support portability for the particular language implementation. However, a language-specific interface is generally only useful for the language in question. MultiSR (see Section 3.2.2.3) uses this latter approach.

In some sense, a goal of Mezcla is to separate thread system policies from thread system mechanisms. However, a more accurate view is that Mezcla attempts to allow a client to *define* the thread policy and to *share the control* of thread mechanisms with the target description. That is, clients help define the resulting thread mechanism.

- **Client Descriptions** In order to separate concerns, we need a way to specify the language-specific functionality. As described in Section 6.1.1, we need a client description language that is both simple and expressive. Unfortunately, these attributes are often at odds with each other. A simple description language may sacrifice the ability to define sufficiently rich thread primitives, while a very expressive language may be overly complex. A challenge, then, is to find the right balance between simplicity and expressiveness.

- **Target Descriptions** In addition to a client description language, separating concerns also requires an approach to specifying target functionality. A target description language will necessarily be more complex than a client description language because different targets can vary greatly in their level of support for threads. In addition, we are intentionally keeping the client description language simple, off-loading complexity onto the target description language. We assume that target descriptions will not be written as often as client descriptions, justifying a complicated, but expressive target language. Our challenge is to minimize the complexity of target descriptions while supporting a high-degree of specialization.

- **Thread Generation** The final ingredient for the framework is a tool that will fuse together a client description and a target description to produce a specialized thread system. The thread generation tool requires knowledge of both the client description language and the target description language as well as a set of rules for combining the descriptions. Therefore, the complexity of the thread generation tool will be highly dependent on the design of the two description languages.

6.2 Concepts

The section introduces two important sets of concepts: the *Mezcla abstract thread model* and the *Mezcla specialization model.* The former model more precisely defines our notion of a thread and thread related structures, while the latter clarifies our notion of thread system specialization. These models will be used to help guide the rest of the design issues addressed in this chapter.

6.2.1 Mezcla Abstract Thread Model

The Mezcla abstract thread model defines our notion of a thread and various thread related structures and operations. The following description formalizes the concept of a *thread* introduced in Chapter 2. The model is *abstract* in the sense that it only makes general definitions and assumptions. It is neither formal nor is it complete; it is intended as a reference to help guide the design of the Mezcla thread framework.

6.2.1.1 Thread Definition

A thread, T, represents an *execution stream.* A thread is *active* when it is executing processor instructions. Threads execute the instructions that make up the code of a program. As such, threads can modify both processor registers and memory. A thread has an associated stack that is used for parameter passing, register spilling, and possibly context saving during thread execution. To distinguish threads, we use the notation T_i, where i is a unique thread identifier.

Unlike a traditional operating system process, a thread often shares memory with other threads that are possibly executing in parallel. As such, threads need to take measures to ensure exclusive access to shared data.

A thread can be in one of several states: *initialized* (IN), *ready* (RD), *running* (RN), *blocked* (BL), and *exited* (EX). A thread is only active in the RN state. In all other states, a thread is a passive entity, represented by data in memory. When a thread is in the RD state, we often refer to it as "runnable". Figure 6.2 illustrates the various thread

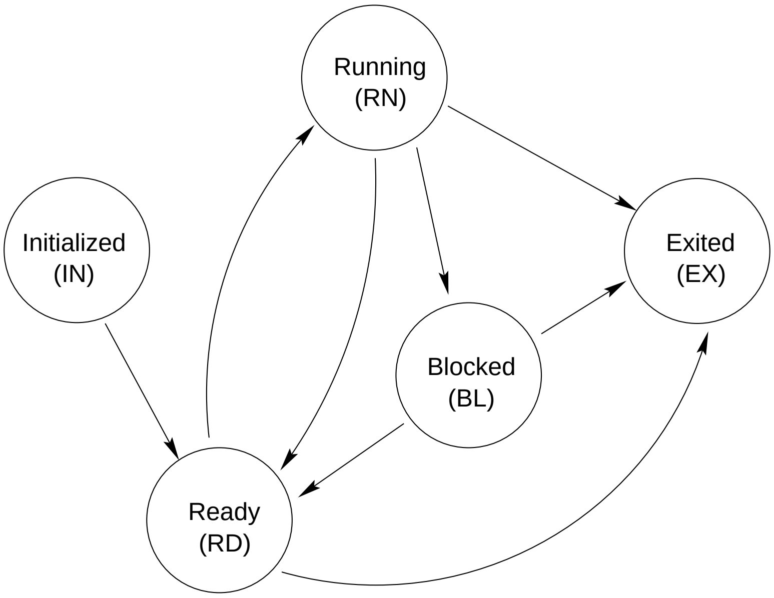

Figure 6.2: Thread States and Transitions

states and typical state transitions.

Threads transition between states in response to different actions. We use the notation $T_i[X \to Y]$ to describe a state transition on thread T_i from state X to state Y. Table 6.1 associates actions with state transitions. An action may affect one or two threads. Create, Activate, and Unblock affect only one thread. Yield, Block, and Exit may affect one or two threads. The Kill action may affect one or two threads depending on the state of the thread being killed. Some actions can be issued by running threads (e.g., Create, Activate, and Unblock); however, some actions can be caused by external events, such as unblocking due to input/output completion (e.g., Unblock).

The Create action produces a new thread by allocating memory for a thread and putting it into the IN state. Note that the Create action does not make the new thread runnable. For that, the Activate action must be used to move a thread from the IN state to the RD state. Some thread systems, such as Pthreads, combine the Create and Activate actions into a single programmer visible function (see `pthread_create()` in Section 2.4).

The Yield action stops the currently running thread and allows a new thread to run. The Yield action may be voluntary (when invoked by a user program) or it may be involuntary (when invoked by a preemptive scheduler). If another thread is runnable (in the RD) state, then the yielding thread is put into the RD state and the runnable thread is put

Action	Description	Transition
Create	Create a new thread	$T_i[\emptyset \rightarrow IN]$
Activate	Make a thread runnable	$T_i[IN \rightarrow RD]$
Yield	Allow a different thread to run	$T_i[RN \rightarrow RD], T_j[RD \rightarrow RN]$ or $T_i[RN \rightarrow RN]$
Block	Block a thread	$T_i[RN \rightarrow BL], T_j[RD \rightarrow RN]$ or $T_i[RN \rightarrow BL]$
Unblock	Unblock a thread	$T_i[BL \rightarrow RD]$
Exit	Exit a thread (terminate)	$T_i[RN \rightarrow EX], T_j[RD \rightarrow RN]$
Kill	Explicitly terminate another thread	$T_i[RN \rightarrow EX], T_j[RD \rightarrow RN]$ or $T_i[\{IN, RD, BL\} \rightarrow EX]$

Table 6.1: Thread Actions and Transitions

into the RN state. If no other threads are runnable, then the yielding thread simply remains in the RN state and continues to execute. Note that a scheduling policy may enforce thread priorities, in which case a yielding thread may not switch to another runnable thread at a lower priority.

The Block action stops the currently running thread, puts it into the BL state, and allows another thread to run (if one is runnable). Conversely, the Unblock action moves a thread from the BL state to the RD state, thus making the unblocked thread runnable.

The Exit action is used by a thread to explicitly terminate itself. It puts the exiting thread into the EX state and allows another thread to run. The Kill action is used by a thread to explicitly terminate another thread. However, the thread to be killed may be in one of four states: RN, IN, RD, and BL. If the thread to be killed is in the RN state, then it is put in the EX state and another thread is allowed to run; otherwise, it is simply put into the EX state. The Kill action can be complicated to implement because the thread to be killed can be in one of many thread states.

Note that for the Block, Yield, Exit, and Kill actions on a multiprocessor, T_j might be T_{idle}, which represents an idle thread. This situation occurs when there are more processors than runnable threads. Also note that the EX is used both for completeness of the model and, in some cases, to support certain thread system implementations. For example, threads may have to be in an explicit EX state to be reclaimed by a garbage collector.

6.2.1.2 Thread Control Blocks (TCBs)

In addition to states and transitions, a thread also has passive components, i.e., data in memory. At a minimum, a thread needs a stack. However, it is common to use a data structure called the thread control block (TCB) to keep track of all the information related to a thread. A TCB is similar to the process control block used in traditional operating system kernels to keep track of processes. The TCB is simply a collection of data, usually as a C struct, that defines the state of a thread. In Mezcla, TCBs and thread stacks are allocated separately, i.e., the TCB is not allocated on the thread stack. The TCB is a fundamental concept in the Mezcla thread model.

A thread is identified and accessed through its corresponding TCB. Therefore, in some contexts, the distinction between a thread and a TCB is not always clear. In many cases, the terms TCB and thread can be used interchangeably. We will make the distinction when it is important.

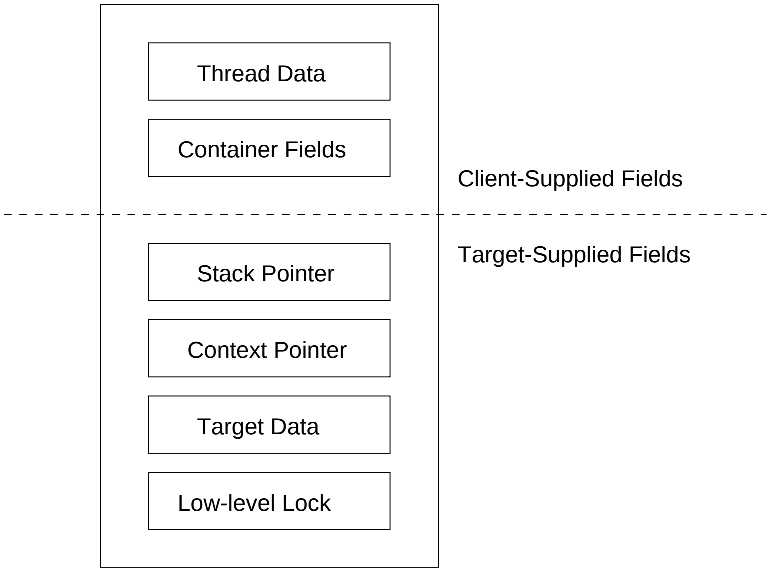

Figure 6.3: The Thread Control Block (TCB)

In Mezcla, we divide the TCB into two parts: the client-supplied fields and the target-supplied fields (see Figure 6.3). The client-supplied fields include any client-specific thread data and TCB container fields (see below). The client-specific fields are used by the client to support language or library specific functionality. The container fields are used to support the linking of TCBs inside a TCB container.

The target-supplied fields include a stack pointer, a context, a low-level lock, and target data. The stack pointer is an address to the location in memory where the stack region begins. The context pointer points to the saved state of a thread that is not running. The low-level lock is used on targets that support true parallelism or preemption. Finally, a TCB will often have target-specific data fields to support particular target functionality.

The TCB is central to supporting client-specific thread semantics and to supporting those semantics on particular targets. One of the objectives of Mezcla is to logically separate client TCB concerns from target TCB concerns so that they can be be specified independently, then combined later to create a single, specialized TCB.

6.2.1.3 Thread Control Block Containers

A TCB container is simply a data structure used to store TCBs in memory. In addition to storing TCBs, containers are also fundamental to *scheduling* threads because TCB containers are usually where TCBs reside when they are not active. That is, when a TCB is in the *ready* or *blocked* state, it is usually in a container. Often, a thread system will employ a single, global ready container for holding ready threads.

Containers support the following operations: **init**, **get**, **put**, and **head**. The **init** operation initializes a container. The **get** operation retrieves a TCB from a container, and the **put** operation places a TCB into a container.

The implementation of a container dictates the order in which TCBs are removed from and placed into a container. Therefore, a TCB container effectively determines how threads are scheduled. For example, when a program invokes the Yield action, a TCB must be selected from a ready container using the **get** operation. Containers can have a variety of implementations, e.g., simple FIFO queues for FCFS scheduling or a priority queue for supporting priority scheduling.

In Mezcla, we leave the definition of TCB containers up to the client. This way a client can employ any desired scheduling policy. Different containers can hold threads in different states.

For example, one might use a priority queue for the ready container and simple FIFO queues for blocked containers. The implementation of a container must be reflected in the client-supplied TCB container fields.

6.2.1.4 Execution Control Blocks (ECBs)

Another important thread system data structure is the execution control block (ECB). The ECB encapsulates processor-specific thread execution information. The ECB holds information that aides in the scheduling of threads on a single processor. Usually, the ECB contains a pointer to the currently running TCB, since a running TCB does not reside in a ready or blocked container. A multiprocessor implementation will associate an ECB with each processor or virtual processor[1] available to the thread system.

6.2.1.5 System Control Blocks (SCBs)

A Mezcla thread system will have at least one system control block (SCB). An SCB provides a global environment for ECBs and TCBs. The SCB stores client supplied callback functions for stack and TCB allocation and deallocation. These pointers could be stored as global variables, but the SCB provides a uniform naming convention and a single allocation block for global data that can be reclaimed by the client once multi-threaded operation has finished. In addition, putting global thread system data in an SCB allows multiple multi-threaded systems to run in the same address space.

6.2.2 Mezcla Specialization Model

We now describe more precisely what is meant by *specialization* and what exactly will be specialized in a Mezcla thread system. To guide the rest of the design, we define the *Mezcla specialization model*.

Our approach to specialization, summarized in Figure 6.4, is to divide the concept of a thread system into two halves: the client half and the Mezcla half. Figure 6.4 illustrates the different parts of a Mezcla-based thread system and the relationship between these parts. The goal is to have the Mezcla half provide low-level thread primitives whose basic functionality is dictated by the client. The client specifies custom thread system information such as the client half of the TCB and TCB containers. A thread generation tool fuses the client-specific thread information with a particular platform target. In some cases, client specific functionality can be achieved with client callbacks. For performance reasons, callbacks should be used only when they are not on the critical path of common thread

[1]Typically, an operating system does not assign real processors to a thread system, rather it assigns kernel-level threads that serve as virtual processors.

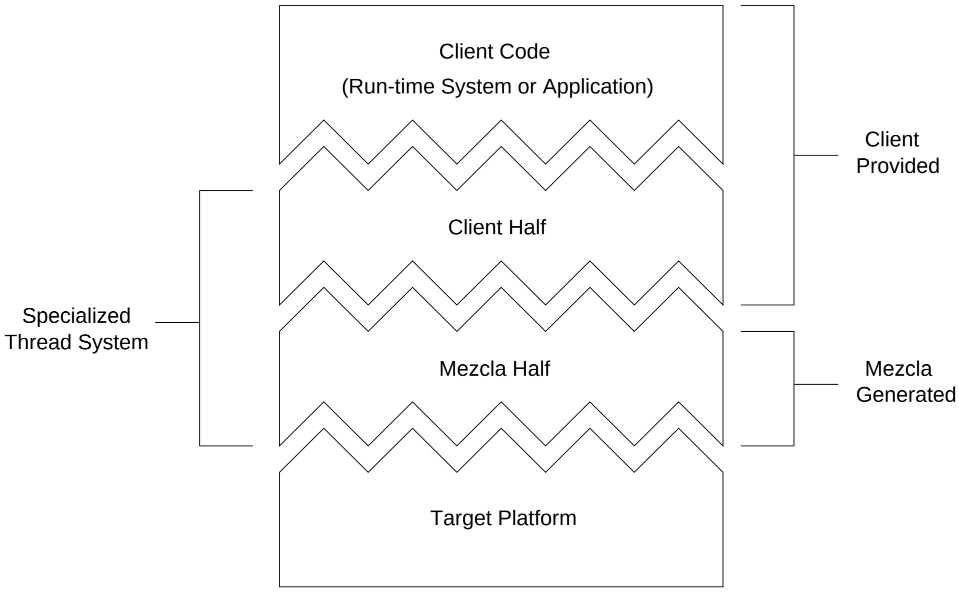

Figure 6.4: Specialization Model

operations or when the overhead of the callback is dominated by the cost of the specific callback operation.

A complete thread system consists of a specialized Mezcla target (which includes platform-specific code required for a target) and the client half. The client half is implemented in terms of the Mezcla primitives. Ultimately, the client half exports thread functionality as run-time system interfaces or library interfaces. These interfaces are used by a client application program.

Given this model of specialization, we have to address four key design issues:

- Determine what primitives will be exported by the Mezcla half.
- Design a client description language that allows a client to easily express how the Mezcla half will be specialized.
- Design a target description language that allows us to capture the thread functionality available on a particular target.
- Devise a method for combining client descriptions and target descriptions to generate a specialized set of Mezcla primitives.

Before addressing these design issues, we characterize the different types of target functionality that we intend to support in Mezcla generated thread systems.

6.3 Target Functionality

Before presenting design issues for the Mezcla primitives, we briefly discuss issues related to different types of targets. The issues addressed in this section will help guide the design of the Mezcla primitives. Recall from the assumptions listed in Section 6.1.2 that Mezcla is primarily intended to help build specialized *user-level* thread systems. As such, this section addresses target issues related to implementing user-level threads. One exception concerns the ability to target a high-level thread library such as Pthreads. Therefore, we consider implementation issues for non-preemptive uniprocessor threads, preemptive uniprocessor threads, non-preemptive multiprocessor threads, and Pthreads. Much of the functionality we describe here comes from the analysis of thread systems given in Chapter 5.

6.3.1 Non-preemptive Uniprocessor Threads

Perhaps the simplest thread target is one that implements non-preemptive uniprocessor threads. For this target, only a single thread can be executing at any given time. Therefore, no low-level locking is required inside the thread system to protect access to the ready container or to blocked containers. In addition, a thread will only switch to another thread *voluntarily* by yielding or blocking. A non-preemptive uniprocessor implementation requires a single ECB and, like all Mezcla thread systems, a single SCB. ECBs and the current TCB can be located via a global pointer.

The main design consideration for non-preemptive uniprocessor threads is the method of context switching: direct switch or preswitch. Section 5.3.2 discussed the relative merits of both methods. We support both methods in Mezcla. Using the direct switch method with a uniprocessor target should have marginally better performance than using the preswitch method.

6.3.2 Preemptive Uniprocessors Threads

A logical, but non-trivial extension to a non-preemptive uniprocessor target is to add preemption. For this target, a thread can be interrupted at any time by a timer interrupt unless measures are taken to prevent the interrupt. An interrupt handler is used to force a thread Yield action, thus possibly allowing another thread to run if one is available. The main problem with preemption is that shared thread system data must now be protected by locks to prevent a context switch in the middle of a critical section. However, a preemptive uniprocessor target still can run only one thread at a time, so critical sections can be protected by disabling timer and input/output interrupts.

6.3.3 Non-preemptive Multiprocessor Threads

A non-preemptive multiprocessor target is more complicated than a uniprocessor target because now two or more threads can be running simultaneously. Low-level, multiprocessor-safe locks, such as spinlocks, are required to ensure mutual exclusion to shared thread system data. A multiprocessor target requires a single SCB and multiple ECBs (one for each processor). In addition, this target type requires a method for associating a processor with a ECB. The ability to find the current ECB is needed to determine the current TCB for a specific processor, as a pointer to the current TCB is typically stored in the ECB. Section 5.3.2 showed that the preswitch context switching method is more appropriate for multiprocessors, so we only consider this method for multiprocessor targets.

A multiprocessor target also offers more complex scheduling choices. In a uniprocessor target, it is customary to have a single, global ready container for runnable threads. Such a scheme is also possible in a multiprocessor target with the exception that the ready container needs to be protected by a lock. Depending on the application, the single ready container could become a source of contention. An alternative approach is to associate a ready container with each ECB. This approach alleviates contention for a single ready container, but complicates scheduling in the event that an ECB runs out of threads and must obtain threads from a different ECB.

6.3.4 Pthreads

Finally, we consider a high-level library target: Pthreads. We would like to map the low-level Mezcla primitives to the high-level functions found in the Pthreads library. This seems counterintuitive, but it has the potential of making Mezcla-based thread systems extremely portable. Our approach is to carefully design the primitives to facilitate this mapping. However, because mapping to Pthreads introduces a level of indirection to thread functionality, we do not expect the Pthreads target to perform well. In fact, we intentionally make design decisions for the Mezcla primitives that favor the user-level thread targets. Such decisions may have a negative performance affect on the Pthreads target. In addition, the user-level thread target allows the client to define a scheduling policy via TCB container operations. Because Pthreads provides its own scheduling, any client supplied scheduling code will be ignored.

6.4 Primitives

The Mezcla specialization model described in Section 6.2.2 divides a thread system into a client half and a Mezcla half. The interface between the two pieces is a set of shared data types and specialized primitives. We now define the types of primitives that should be exported by the Mezcla half. This section examines several possible primitives to include in a Mezcla implementation. The design of the primitives is guided by the following principles:

- **Uniform treatment of all target types** The Mezcla primitives should be designed such that they can be mapped to all the targets outlined in Section 6.3. They should favor designs that will perform better for the user-level thread targets than the Pthreads target.

- **Thin primitives** The level of abstraction of the Mezcla primitives should be as low as possible without exposing target-specific details.

- **Client control over data allocation** The Mezcla primitives should give the client control over allocation and deallocation of thread-related data objects, such as TCBs and stacks. This allows a client to perform certain optimizations such as TCB and stack caching.

- **Client control over thread system locks** To avoid the duplication of locking described in Section 5.3.1, the Mezcla primitives should give the client as much control as possible over what thread data is locked and when the locking occurs.

- **Client-specific blocking and unblocking** Fundamental to most high-level synchronization mechanisms found in language run-time systems or thread libraries is the notion of *blocking* and *unblocking* threads. The Mezcla primitives should allow the client to define multiple, custom versions of these primitives that are used to implement different types of client-specific synchronization mechanisms.

The fundamental data types shared between the client half and the Mezcla half are TCBs and TCB containers. As such, the Mezcla primitives are designed to work with any client-defined versions of these data types.

This section examines both client-supplied callback operations and the Mezcla primitives. We divide the Mezcla primitives into two types of primitives: *fixed* and *custom*. The fixed primitives will exist for every Mezcla client, whereas the custom primitives are defined by the client. Both fixed and custom primitives will be *specialized* for a particular client/target pair during thread generation. Despite the term *fixed*, even some of the fixed primitives will be modified to support the thread data types specified by a client.

6.4.1 Client-supplied Callbacks

As defined in the Mezcla specialization model (see Section 6.2.2), some client-specific functionality can be implemented using callbacks. Recall that a callback should be used only when the callback is not on the critical path of common thread operations or when the overhead of the callback is dominated by the cost of the specific callback operation. In a Mezcla thread system, we use callbacks for client-specific memory allocation and deallocation operations. Using callbacks for these operations allows the client to have control over the allocation of thread related data objects. Perhaps the most important thread objects to be client controlled are the TCB and the thread stack, although the other thread structures — TCB containers, ECBs, and SCBs — are also candidates for client control.

6.4.2 Fixed Primitives

6.4.2.1 System Initialization

All thread systems need a method of initialization to set up internal thread system data structures and target-specific functionality. For a preemption target, initialization is required to install a timer interrupt handler. For multiprocessors, initialization is required to startup virtual processors and perhaps to initialize low-level locks.

Two common forms of initialization are *explicit* and *implicit*. Explicit initialization requires a call to a thread system initialization function before multi-threaded execution can begin, while implicit initialization occurs without direct intervention by a programmer. (The thread system is initialized either right at program load time or dynamically during program execution.)

Because we allow Mezcla clients to install callback operations for thread data allocation, it is easiest to use explicit initialization. The initialization primitive can be passed pointers to any necessary callback functions.

6.4.2.2 Low-level Locks

To support the mutual exclusion requirements of the preemption and multiprocessor targets, the Mezcla primitives must provide support for locking. Unfortunately, locking is not needed for non-preemptive uniprocessor targets. However, to satisfy the design principle of uniform treatment of target types, we require clients to use locks. In the case that locks are not needed, the locking primitives can simply be ignored. The locking primitives should be able to accommodate both preemption locks (i.e., disabling interrupts) and multiprocessor locks.

6.4.2.3 Thread Initialization and Activation

The Mezcla abstract thread model defines actions for thread initialization and thread activation, which can be combined or separate. The Mezcla primitives should support the separate approach to give the client maximum flexibility.

An initialization primitive should allow the client to first allocate a TCB and initialize any client-supplied TCB fields. In addition to a TCB pointer, the initialization primitive should take a pointer to a thread start function and an argument to be passed to that function. The initialization primitive should also take care of setting the target-supplied fields and should be specialized to accommodate the client-defined TCB data type.

An activation primitive will take an initialized TCB and put it into a global ready container using the client-supplied container operations. Note that separate initialization and activation primitives are natural for the user-level thread targets, but not for the Pthreads target. In this example, extra coding is needed in the Pthreads target to support certain Mezcla primitives. For Pthreads, initialization simply sets a function pointer and argument value in the Pthreads portion of the TCB. The actual `pthread_create()` will not occur until the activation primitive is invoked.

6.4.2.4 Thread Yielding

Because Mezcla supports non-preemptive, uniprocessor targets, it is important to provide a yield primitive. The yield primitive will attempt to retrieve a thread from the ready container. If one exists, the currently running thread will be placed into the ready container and the new thread will be allowed to run. The yield primitive must use a target supplied context switching method. In the case of Pthreads, the yield primitive can map directly to the Pthreads yield function.

6.4.2.5 Thread Exiting

An exit primitive is needed to allow a thread to explicitly terminate itself. Threads may also exit by simply returning from the thread start function. When a thread exits, there may not be any runnable threads. When this situation occurs on a uniprocessor target, the thread system can simply exit, as without any runnable threads the program can no longer make progress. However, on a multiprocessor target, if no threads are available, an idle thread should be scheduled in the event that there are threads running on other processors. The idle thread will periodically check to see if new or unblocked threads are ready to run until the program explicitly terminates.

6.4.2.6 Thread Identification

The client half of a Mezcla thread system needs to be able to access client-specific TCB fields of the currently executing threads. Therefore, a primitive is required to provide a client with a pointer to the current thread.

6.4.2.7 Thread Killing

A thread kill primitive allows a thread to explicitly kill another thread. Such a primitive introduces several complexities for both the thread system and client applications that use the thread system. The most difficult problem to deal with is the situation where one thread attempts to kill another thread holding onto one or more locks. The question is, what should be done with the locks? There is not a clear answer to this question. We currently do not support explicit thread killing in Mezcla.

6.4.3 Custom Primitives

A Mezcla thread system also needs to support client-specific block and unblock primitives so a client can build custom synchronization mechanisms.

6.4.3.1 Blocking Threads

Mezcla needs to be able to generate multiple, client-specific blocking primitives. A blocking primitive will perform the Block action and corresponding transitions as described in the Mezcla abstract thread model (see Section 6.2.1). Two approaches for blocking threads are *coupled* and *decoupled*.

In the coupled approach, the Block action is achieved by a single `block` primitive. That is, the block transition and the yield transition are combined into one step — the most common way to implement the Block action.

In the decoupled approach, the Block action is viewed as two steps. First, the programmer specifies where the current thread should block (it places the current thread in a blocked container). Then, the programmer explicitly transfers execution to another thread using a yield primitive. The decoupled approach is more complicated to use, but some language run-time systems (such as SR's) use this approach.

A Mezcla implementation should support at least one of these two approaches.

6.4.3.2 Unblocking Threads

Mezcla also needs to support the generation of custom unblocking primitives. Compared with blocking primitives, unblocking primitives are relatively simple; they need only to perform an Unblock action on a single thread. This usually involves removing a thread from a blocked container, and then putting it into the ready container.

6.5 Thread Generation

This section discusses what is needed to build a thread generation tool: a client description language for specifying language-specific thread requirements, a target description language for encoding the functionality of a specific platform, and a tool for composing client descriptions with target descriptions. The discussion is given in terms of the basic concepts presented in Section 6.2 and the primitives presented in Section 6.4.

6.5.1 Client Description Language

One of the Mezcla design goals described in Section 6.1.1 is to provide portability through a client description language. This language should allow the client to specify any client specific concerns over the implementation of the thread primitives outlined in Section 6.4. A description should accommodate the specification of both client-specific data and client-specific code. Its data specification should include:

- Thread Control Blocks (TCBs)
- Thread Control Block Containers

The code specification should allow the client to specialize the mechanics of the different Mezcla primitives. A code specification should include:

- Scheduling Operations
- Synchronization Primitives

6.5.2 Target Description Language

A target description language is needed to codify the functionality of the different targets presented in Section 6.3. Representing target functionality is a difficult problem because different targets can have vastly different implementation requirements. The target description language needs to be able to support the specialization of the primitives from Section 6.4 in terms of the client information supplied in the client description language.

6.5.3 Thread Generation Tool

Given client and target descriptions, a thread generation tool is needed to combine these descriptions together to produce specialized versions of the Mezcla primitives. The thread generation tool creates the specialized Mezcla half of a thread system as defined by the Mezcla specialization model (see Section 6.2.2). The implementation of a thread generation tool will be largely dictated by the format of the client and target description languages.

6.5.4 Implementation Alternatives

This section briefly outlines two different implementation alternatives for thread generation: a template approach based on macros and a compiler approach.

6.5.4.1 Template Approach

A simple implementation approach would describe targets as C code templates that are filled in with client-defined C macros. A great deal of specialization can be achieved with judicious use of the C macro preprocessor. Using macros, the client description language would be simply a C source file and a client would define values for the "client-required" macro name. Macros are quite flexible; they can be used to specify both data and function declarations. For example, a client could provide client-specific TCB fields using a simple macro, which would be joined with the target TCB fields to create a joined TCB structure.

This approach requires very little infrastructure in the way of client and target description parsing as well as the thread generation tool itself. In fact, thread generation is reduced to simply using the C compiler.

Unfortunately, writing macros can be tedious. Perhaps more problematic is that macros can perform only limited types of code generation and specialization based on direct substitution. A macro can only be expanded. It can not rearrange or generate C code based on the values of the macros or the context in which the macro is being expanded. Furthermore, writing correct macros is difficult because no type checking is performed by the compiler on macro definitions.

6.5.4.2 Compiler Approach

A more sophisticated approach to generating thread systems is to define a "real" programming language for describing client specific thread concerns. By defining a language, a thread generation tool would serve as a compiler and could perform standard syntax and type checking on a client description "program". Furthermore, using a compiler, the client description could be transformed into an intermediate representation that is more suited to combining with a target description. Given an intermediate representation, it might even be possible to perform target-independent optimizations on the client description. A compiler approach can offer much more flexibility over the template approach in how code can be generated since it is not limited to simple substitution.

In the compiler approach, we still need a way to describe target functionality. One approach, similar to the macro-based template approach, could be to employ targets as "code templates" that are suitable for accepting an intermediate form of the client description. Ultimately, it would be nice to have a target description language that encodes target functionality as concisely as possible so that target descriptions are easy to write and maintain.

6.6 Frameworks in Other Domains

Frameworks in other system programming domains, including Lex and Yacc for compilers, the *x*-kernel for network protocols, and the Flux OS Toolkit for operating systems, have inspired the design of the Mezcla thread framework. These frameworks take different approaches to enabling programmers to easily create custom, domain-specific system-level software. This section briefly summarizes each of these frameworks and concludes by discussing how these frameworks have influenced Mezcla.

6.6.1 Lex and Yacc

Lex [76] and Yacc [66] are programs used to generate lexical scanners and language parsers, respectively. They are programs used to create programs or, more specifically, useful application-specific libraries. Used together, these programs facilitate the development of compilers, interpreters, and any program that must understand an input language. While Lex and Yacc are not commonly referred to as a "framework", together, they do fit the notion of a framework as it is used in this dissertation[2]. Both Lex and Yacc have specification languages, a tool to generate C code from a specification, and conventions for using the resulting C code. In fact, Lex and Yacc share a common convention that allows Lex scanners to work naturally with Yacc parsers.

Lex and Yacc have relatively simple specification languages. The Lex specification language allows a programmer to express how to identify tokens from a textual input stream using regular expressions and to associate code fragments, written in C, with different token specifications. The Lex tool takes this specification as input and generates a program that will identify the specified tokens and execute the corresponding code fragment. The resulting

[2]Lex and Yacc are more commonly referred to as programming tools that are standard in most versions of UNIX.

"scanner" can be used by a larger program that needs to scan text for tokens or it can be used directly by a Yacc generated parser.

Lex has several desirable qualities. First, as mentioned previously, it has a simple and concise specification language. Second, by allowing code fragments to be associated with a regular expression, Lex gives the programmer complete flexibility in determining what to do when a token is matched. Third, in addition to associating C code fragments with each token specifier, Lex also allows the C code to "take control" of the input stream if it is necessary to process the input in ways impossible or not easy to express using Lex regular expressions. For example, it is common to use C code to quickly match C-style comments, rather than to rely on what would be a complicated regular expression. Finally, the output of Lex is C code, which can be readily used by other C programs. The generated C code is portable and can be used on most platforms that provide a C compiler.

The C fragments must obey certain Lex conventions so that correct scanners are generated. This approach provides a high degree of flexibility, but it also has two significant drawbacks. First, Lex assumes that the programmer provides properly written code fragments. Therefore, there is no way for Lex to ensure that the resulting scanner will work as expected. Second, Lex does not itself perform type checking and semantic analysis of the associated, user-supplied code; that code is checked only later when the entire scanner code is compiled by a C compiler. To catch problems, the programmer must debug the generated scanner.

The Yacc specification language is similar is spirit to the Lex specification language. It allows a programmer to specify a language grammar. The major construct in the specification language is a *rule*, which is used to identify a structured input. A programmer can associate C code with each rule. Similar to Lex, the C code must obey certain Yacc conventions. Yacc also shares the advantages and disadvantages associated with Lex.

Lex and Yacc have not only influenced the design of Mezcla, but they also heavily influenced the prototype implementation of Mezcla presented in Chapter 8.

6.6.2 The x-kernel

The x-kernel [62] is an object-oriented framework for building network protocols. Specifically, the x-kernel can be used to construct custom protocol stacks that can be used by an operating system or an application program. The lowest level of the protocol stack communicates directly with a network interface controller. The highest level of the protocol stack provides an API to be used by an application or to be exported by an operating system. The x-kernel was used to provide a TCP/IP protocol stack for the OSKit implementation of SR described in Section 4.4.

The x-kernel framework consists of four major components: an interface specification for protocol and session objects, a protocol graph specification language, a protocol composition tool, and a rich set of supporting libraries that assist in the implementation of protocol objects.

Perhaps the most fundamental components of the x-kernel are the protocol and session objects. Protocol objects define specific protocols (such as UDP and TCP). Session objects maintain connection instances and any state associated with any particular connection. As described in [90], protocol objects export operations for opening channels—resulting in the creation of a session object—and session objects export operations for sending and receiving messages. Protocols and sessions are rigidly defined using the *uniform protocol interface*.

Once protocol and session objects have been defined, a programmer defines a protocol stack using a protocol graph specification. The specification format is relatively simple; it resembles the dependency format used by the UNIX *make* program. A protocol composer tool reads a graph specification file and generates code that will dynamically configure a protocol stack by properly connecting protocol objects.

The final component of the x-kernel framework is a set of support libraries used to write protocol objects. These libraries include support for messages, participants, events, maps (flexible hash tables), and threads.

Compared to Lex and Yacc, the x-kernel framework is quite different and has several useful qualities. For example, protocols are programmed as independent C modules. The uniform protocol interface defines the structure of these C modules, whereas the structure of C code fragments for Lex and Yacc is free-form and must follow only a few conventions. The x-kernel framework also provides a fairly comprehensive support library for writing protocols, while Lex and Yacc have very few support functions.

Similar to Lex and Yacc, individual x-kernel protocol objects are composed using a protocol graph specification and a composer tool. However, the nature of the composer tool is substantially different from the Lex and Yacc tools. It generates code that will dynamically construct a protocol stack according to the graph specification at run time.

While the x-kernel framework has proven to be very effective for building custom protocols and protocol stacks, it does have some disadvantages as a systems programming framework. First, while the rigid uniform protocol interface is useful for defining a common interface between protocol objects, it can also be overly restrictive. Protocol objects must communicate in well-defined ways. Two protocol objects are not free to *define* the manner in which they communicate. New methods of interfacing must be supported directly by the framework and the uniform protocol interface. Second, the x-kernel framework is rather complicated, perhaps rightly so given that network protocols are inherently complicated. Third, the resulting protocols invoke each other through indirect operations. That is, similar to virtual methods in C++, each operation invocation must first dereference an operation pointer. A more sophisticated tool could possibly generate protocol stacks that avoid some of this indirection.

6.6.3 The Flux OS Toolkit

The Flux OS Toolkit [43], also called the OSKit, is a framework for building operating systems. The OSKit was briefly described in Section 4.4. The OSKit is essentially a collection of useful libraries that encapsulate different aspects of OS functionality. For example, the OSKit provides low-level libraries for boot loading, kernel-mode processor management, memory management, debugging, device drivers, and multiprocessor support. The higher-level libraries include a minimal C library, threads, file systems, and network protocols. Many of the libraries have been imported from existing operating systems, including Linux and FreeBSD. Extensive documentation is provided for each of the libraries in order to expose the implementation details so that the programmer knows exactly how they work.

The OSKit user can use just a few of the libraries and implement the rest of the higher-level functionality, or start with several of the higher-level libraries and incrementally replace libraries as custom versions are written. The motivation behind the OSKit is to enable a programmer to create working operating systems as quickly as possible and to facilitate the implementation of new operating system designs.

The conventions for combining libraries is largely undefined. Some libraries do have some conventions and expect a particular *execution environment.* This framework gives the programmer a large degree of flexibility in building custom operating systems.

However, because of this flexibility, the OSKit framework does not provide any tools for automatically generating operating system code. It is completely up to the programmer to provide the necessary glue code to create a functioning operating system.

Recent versions of the OSKit provide modules in the form of COM (Component Object Model) [83] objects. The ultimate goal is to define almost all OSKit libraries as COM objects, thus providing a more structured environment for composing different libraries.

Compared to Lex and Yacc and the *x*-kernel, the OSKit is very *unconstrained.* As such, the burden of composition is largely left to the programmer. The OSKit has a few loose conventions used by some of the libraries. With the exception of COM-based components, most of the OSKit components are simple function libraries. This approach provides a great deal of flexibility at the expense of a large programmer burden. Just as network protocols are inherently complex, so are operating systems; therefore, the nature of the domain may require a complicated, but flexible framework.

6.6.4 Discussion

The preceding sections presented three programming frameworks for different systems domains: Lex and Yacc, the *x*-kernel, and the OSKit. We have examined these frameworks to help guide the design the Mezcla thread framework. This section briefly discusses the relative merits of the different frameworks and what aspects of the frameworks we incorporated into Mezcla.

Each of the frameworks provides varying degrees of structure. The *x*-kernel provides the highest degree of structure, while Lex and Yacc are less structured and the OSKit has very little structure. There is an apparent relationship between the degree of structure and degree of automatic code generation: the higher the degree of structure, the easier it is to automate code generation and composition. Of course, as the previous sections indicate, the particular domain plays a significant role in how much structure is needed. It seems that a good systems-level programming framework seeks to match a particular domain with the right amount of structure to achieve the desired level of automatic code generation.

The frameworks ensure varying degrees of correctness in their resulting programs. The *x*-kernel enforces protocol object interaction through well-defined interfaces. While an *x*-kernel programmer is free to write incorrect protocol code, the protocol composer can at least ensure that each protocol object exports all the necessary interfaces. Correctness, while not enforced, is partially facilitated by meeting the requirements of the interfaces. On the other hand, the Lex and Yacc framework and the OSKit framework rely on *programming conventions* to produce correct generated code. In this case, the programmer carries a greater burden in ensuring program correctness.

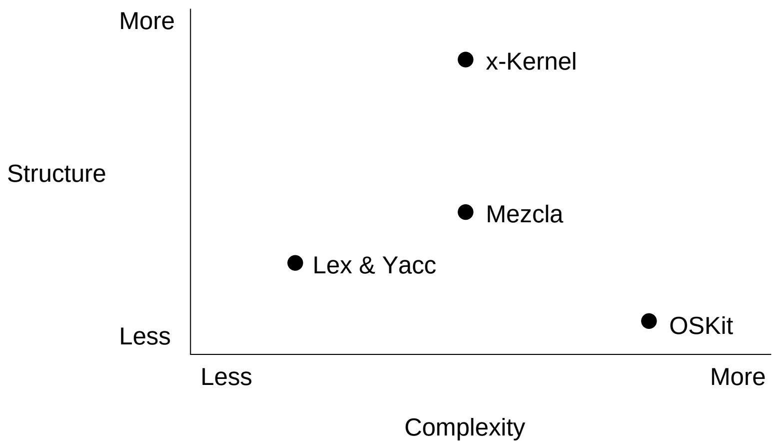

Figure 6.5: The Relationship Between Mezcla and Other Frameworks

The frameworks also have varying degrees of complexity. Often, this complexity is a function of a particular domain. However, it is interesting to note that the degree of complexity does not directly correlate with the degree of structure. Figure 6.5 provides a graph that shows how we view the relationship of the Mezcla thread framework to the other frameworks presented in this section. The x-kernel is highly structured, but is also very complex. Lex and Yacc are less structured, but relatively simple. Finally, the OSKit is highly unstructured, but very complex.

The domain of thread systems is perhaps more complex than that of scanners and parsers, but less complex than that of operating systems. It also seems that thread systems are perhaps as complex as network protocols, but they do not share the same amount of structure. Therefore, based on the analysis of the different frameworks presented in this section, we want to achieve the simplicity of Lex and Yacc, impose a degree of structure close to that of the x-kernel, and enable the flexibility of the OSKit. Clearly, there is tension in trying to meet all of these requirements. The design of Mezcla has been guided by the goal of balancing these requirements. However, the actual level of simplicity, structure, and flexibility largely depends on a particular implementation of Mezcla.

6.7 Summary

This chapter described the Mezcla Abstract thread model and the Mezcla specialization model. These models are useful in defining all aspects of the framework: the basic primitives, the client description language, the target description language, and the thread generation tool. Detailed design options and requirements were considered and presented for each of these aspects.

To help guide the design of Mezcla, we examined three other frameworks for systems programming domains: Lex and Yacc for compilers, the x-kernel for network protocols, and the OSKit for operating system kernel. Each of these frameworks takes a different approach to helping programmers build system-level software. We found that a good systems framework must establish the right balance between structure and flexibility for a particular problem domain.

The next two chapters explore a prototype implementation of the Mezcla thread framework. Chapter 7 shows how to build a simple thread system using the prototype and Chapter 8 gives a detailed presentation of the prototype implementation.

Chapter 7

The Mezcla Thread Framework: Prototype Example

This chapter gives a brief overview and a simple example of how to use the Mezcla thread framework prototype. The overview gives the *big picture* for using Mezcla to build thread systems. The example shows how to implement a simple thread system, called `MutexThreads`, using the Mezcla framework. The details of the Mezcla prototype are given in Chapter 8.

7.1 Overview

Building a Mezcla specialized thread system requires the following steps:

1. **Design a client thread interface** Before building a Mezcla-based thread system, a programmer must define the kinds of thread functions that will be needed in the client system[1]. In addition, a programmer needs to determine the type of thread scheduling required by the client.

2. **Write a client description file** In order to define some client-specific aspects of the new thread system, a programmer must write a Mezcla client description file, which must have a `.mtf` extension and follow Mezcla client description rules (detailed in

[1]A *client* is the program that requires or exports thread functionality, such as a concurrent run-time system or a thread library.

Section 8.2). This file specifies client-specific naming conventions, TCB fields, TCB containers, scheduling routines, and custom synchronization primitives.

3. **Generate specialized Mezcla primitives** After writing a client description file, the `tgen` tool is used to generate specialized Mezcla primitives. The `tgen` program is given a client description file and a target selection. It generates as output a client/target specific version of the primitives. The output of `tgen` is a C file and corresponding header file, whose names are determined by the programmer in the client specification file.

4. **Write a thread system based on Mezcla primitives** Based on the Mezcla primitives (detailed in Section 8.1) generated in Step 3, a programmer must implement the client half of the thread system. That is, the client interface defined in Step 1 must be implemented with the Mezcla specialized primitives.

5. **Compile the thread system** Given the generated source code from Step 4, a programmer can now build a complete thread system. This step usually involves compiling the client-half source and the generated source to produce object code. The resulting object code must be linked with some Mezcla libraries that are required by the generated code.

It is important to keep in mind that the `tgen` program does not generate a *complete* thread system. Rather, it generates specialized primitives that are used by a programmer to build a complete thread system. Also, the development of a real thread system often involves several iterations of the steps given above.

7.2 Simple Example: MutexThreads

MutexThreads is a very simple thread system based on mutex variables. The mutex variable data type and related functions are very similar to those found in Pthreads (see Section 2.4). In addition to providing synchronization with mutex variables, the MutexThreads interface provides functions for creating and exiting threads. Appendix B contains the complete source code for MutexThreads.

While MutexThreads is relatively simple, building a complete thread system involves quite a bit of detail. Figure 7.1 provides a flowchart that shows the steps involved in

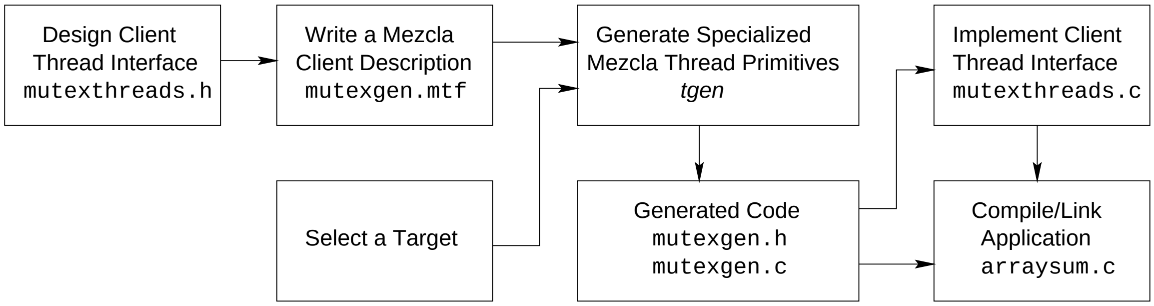

Figure 7.1: MutexThreads Development Steps

developing a Mezcla-based MutexThreads implementation and how to compile and link an application, `arraysum`, that uses MutexThreads.

7.2.1 Interface

Defining the MutexThreads interface corresponds to Step 1 from Section 7.1. The interface is very simple; it provides only functions for initialization and thread creation, termination, and synchronization. Figure 7.2 lists the one MutexThreads data type and all of the MutexThreads functions.

The `mutex_start()` function is used to start multi-threaded execution; it takes a pointer to a start function for the initial thread and an argument to be passed to this function. A call to `mutex_start()` does not return. Threads are created with `mutex_create()`, which takes a function pointer and an argument to be passed to that function. Threads must exit with `mutex_exit()`.

The remaining four functions are used in conjunction with mutex variables as defined by the `mutex` data type. The `mutex` data type is a structure that holds: the mutex value, `value`; a Mezcla lock, `lock`; and a mutex container, `container`. The `lock` variable is required in order to provide mutual exclusion to `mutex` objects in the presence of preemption and true parallelism. The `container` variable is used to hold blocking threads awaiting access to the mutex variable.

The `mutex_init()` and `mutex_destroy()` functions are used to initialize and deallocate a mutex, respectively. The `mutex_lock()` and `mutex_unlock()` functions acquire and release a mutex lock, respectively. Note that `mutex_lock()` is a blocking lock. That is, if the lock cannot be acquired, then the locking thread will block on a queue until the lock

```
#include "mutexgen.h"

/* data types */

typedef struct mutex {
        int value;
        mtf_lock lock;
        mutex_container container;
} mutex;

/* functions */

void mutex_start(mtf_tcb_func *func, void *arg);

void mutex_create(mtf_tcb_func *func, void *arg);
void mutex_exit(void);

void mutex_init(mutex *m);
void mutex_destroy(mutex *m);
void mutex_lock(mutex *m);
void mutex_unlock(mutex *m);
```

Figure 7.2: The MutexThreads Interface (`mutexthreads.h`)

becomes available.

7.2.2 Example Program: `arraysum`

To demonstrate how to use the MutexThreads interface, we provide a simple application, `arraysum`, which implements parallel array summation. The `arraysum` program divides an array of integers into equally sized subarrays. For each subarray, a thread is created to compute the sum of the subarray's elements. The code for the `arraysum` program is listed in Figures 7.3 and 7.4.

Figure 7.3 lists the startup code for `arraysum`. This startup sequence is typical for Mezcla-based thread systems. Program execution begins with a `main()` function, just as in a normal C program. The `main()` function first parses command line arguments and then initiates multi-threaded execution with `mutex_start()`.

The `mutex_start()` function is passed a pointer to an initial thread function and an argument to be passed to that function. The initial thread function, `mutex_main()`, allocates and initializes the array to be summed, `garray`. A loop starts `N` threads using `mutex_create()`. The thread executing `mutex_main()` exits with `mtf_tcb_exit()`.

Each worker thread, when created, is passed the `sum()` function listed in Figure 7.4. The `sum()` function figures out which subarray to sum and then performs the sum calculation. A simple barrier is used to synchronize the worker threads. A mutex variable, `CS`, is used to implement the barrier. Once a thread computes the sum of its subarray, it decrements a global counter, `gbarrier`. If `gbarrier` is 0, then the thread will calculate the total sum of `garray` by summing the totals of each subarray, output the total, and exit; otherwise, it will simply exit. When all the threads have exited, the `arraysum` program terminates.

7.2.3 Client Description

To implement the MutexThreads interface listed in Figure 7.2, a programmer must write the "client half" of the thread system in terms of Mezcla primitives. The Mezcla primitives are specialized for the client based on a specification provided in a client description file. Before describing the client half of the implementation, we show the MutexThreads client description file and the resulting Mezcla primitives. This procedure corresponds to Steps 2 and 3 from Section 7.1.

```
#include <stdio.h>
#include <stdlib.h>
#include "mutexthreads.h"

#define N 2

mutex CS;
int gsize, gchunk, gsum = 0, gbarrier = N;
int *garray;
int gsubarray[N];

void mutex_main(void *arg)
{
        int i;

        /* initialize critical section mutex */
        mutex_init(&CS);

        garray = (int *) mtf_malloc (gsize * (sizeof(int)));
        gchunk = gsize / N;

        /* initialize the global array */
        for (i = 0; i < gsize; i++) {
                garray[i] = i;
        }

        /* initialize the global subarray and start threads */
        for (i = 0; i < N; i++) {
                gsubarray[i] = 0;
                mutex_create((mtf_tcb_func *) sum, (void *) i);
        }

        mtf_tcb_exit();
}

int main(int argc, char **argv)
{
        gsize = atoi(argv[1]);

        mutex_start((mtf_tcb_func *) mutex_main, (void *) 0);
}
```

Figure 7.3: Parallel Array Summation in MutexThreads, Part I

```
void sum(void *arg)
{
        int i;
        int id    = (int) arg;
        int start = id * gchunk;
        int end   = (id + 1) * gchunk;

        /* compute subarray sum */

        for (i = start; i < end; i++) {
                gsubarray[id] += garray[i];
        }

        /* synchronize with other threads */

        mutex_lock(&CS);
        gbarrier--;
        if (gbarrier == 0) {
                for (i = 0; i < N; i++) {
                        gsum += gsubarray[i];
                }
                printf("sum = %d\n", gsum);
        }

        mutex_unlock(&CS);
        mutex_exit();
}
```

Figure 7.4: Parallel Array Summation in MutexThreads, Part II

```
#
# mutexgen.mtf
#
# Mezcla client description file for MutexThreads

# Preamble

set name mutexgen
set client_inc mutexclient.h

# Data Descriptions

data tcb mutex_tcb {
        struct mutex_tcb *next;
        unsigned stacksize;
        mtf_lock lock;
}

data tcb_container mutex_container {
        struct mutex_tcb *head;
        struct mutex_tcb *tail;
}

# Code

code sched mutex_container {
        set init mutex_container_init
        set put  mutex_tcb_put
        set get  mutex_tcb_get
        set head mutex_tcb_head
}

codel sync mutex_container {
        block mutex_block {
                mutex_tcb_put(b_tcb, b_container);
        }
        unblock mutex_unblock {
                b_tcb = mutex_tcb_get(b_container);
        }
}
```

Figure 7.5: MutexThreads Client Description File

MutexThreads requires a relatively simple client description file, which is listed in Figure 7.5. All client description files have three sections: a preamble, a data section, and a code section.

The preamble is used to specify the prefix for the file names of the generated source code and a client-specific include file that is incorporated into the generated code. In Figure 7.5, `mutexgen` is the prefix for generated code, so the output of the thread generation tool (`tgen`) will be `mutexgen.h` and `mutexgen.c`. The client-specific include file is used to provide the generated code access to client-specific TCB container operations. In Figure 7.5, the client-specific include file is called `mutexclient.h`.

The data section is used to define client-specific data structures for the resulting thread system. These data structures include TCBs and TCB containers. MutexThreads has a very simple TCB and TCB container. The TCB consists of a link pointer for putting TCBs into TCB containers, a low-level lock, and a stack size field. The TCB container consists of head and tail pointers for a simple linked list. In this example, `mutex_tcb` is the name of the TCB data type and `mutex_container` is the name of the TCB container data type. The `mutex_tcb` and `mutex_container` data types are used as the basis for the MutexThreads container operations defined in `mutex_client.h` (see Section B.1.2).

Finally, the code section is used to specify client specific scheduling operations and to define custom synchronization primitives. The `code sched` statement defines the basic scheduling operations to be used by the Mezcla generated thread system. The names defined in a `code sched` statement give the generated code access to client-specific scheduling operations, which are used in the generated code each time a TCB is retrieved or stored into a TCB container.

The `codel sync` statement defines a pair of custom synchronization primitives: `mutex_block()` and `mutex_unblock()`. These custom primitives are used to implement `mutex_lock()` and `mutex_unlock()`. The `codel sync` statement allows a programmer to specify the names of the resulting primitive — `mutex_block()` and `mutex_unblock()` — and to associate client-specific blocking code with each operation. By convention, the client-specific code assumes an execution scope that includes two local variables, `b_tcb` and `b_container`. For the `block` operation, `b_tcb` points to the blocking thread and `b_container` is a client-supplied container (determined at run time) in which the `b_tcb` thread should be stored. For the `unblock` operation, `b_container` is the programmer-supplied container from which a thread should be unblocked. The `b_tcb` variable should be set to the TCB retrieved from `b_container`.

The client description file is passed to the thread generation program, `tgen`, to generate specialized versions of the client data types and the Mezcla primitives and to create any custom primitives specified by the client. Figure 7.6 lists the Mezcla primitives generated for MutexThreads.

The Mezcla primitives include multi-threaded initialization, low-level locking, TCB operations, memory allocation, and client-specific custom primitives. Some of the prototypes for the Mezcla primitives are fixed, while others are client specific. A fixed primitive is one that is available to all Mezcla clients. For example, `mtf_init()` and the `mtf_lock_*` functions are fixed, whereas `mtf_tcb_init()` and the custom primitives, `mutex_block()` and `mutex_unblock()`, are client specific. These primitives are the entry points into the *Mezcla half* of the thread system implementation; each is specialized to use the data and code specifications provided in the client description file. The next section shows how to implement the client half of MutexThreads using the Mezcla generated primitives.

7.2.4 Implementation

After writing a proper client specification, a programmer must implement the client thread interfaces (Step 3 from Section 7.1). Specifically, the client half of the thread system must implement the MutexThreads functions listed in Figure 7.2 in terms of Mezcla primitives listed in Figure 7.6. Rather than list the entire implementation, we focus on three MezclaThreads functions: `mutex_create()`, `mutex_lock()`, and `mutex_unlock()`. See Appendix B for the complete source code to MutexThreads.

Figure 7.7 lists the code that implements `mutex_create()`, the MutexThreads function for creating a new thread. A Mezcla-based thread create function must allocate a TCB, set the stack size, and initialize any client-specified TCB fields. The `mtf_tcb_init()` function is used to perform target-specific thread system initialization, which usually consists of establishing a low-level context for the new thread. The `mutex_create()` function also increments a global counter, `mutex_tcount()`, that keeps track of the number of active threads. This counter is used in `mutex_exit()` to determine when the last thread exits so the entire program can terminate.

Figure 7.8 lists the code for `mutex_lock()`. The code checks the current value of the mutex, `m`. If `m->value` $= 0$, then the mutex is available and it is set to 1. If `m->value` $\neq 0$,

```
/* data types */
typedef struct mutex_tcb mutex_tcb;
typedef struct mutex_container mutex_container;

/* initialization and system functions */

int  mtf_init(mtf_tcb_alloc_func *tcb_alloc,
              mtf_tcb_free_func *tcb_free,
              mtf_stack_alloc_func *stack_alloc,
              mtf_stack_free_func *stack_free);

int  mtf_start(mutex_tcb *main_tcb, mtf_tcb_func *main_func,
               void *main_arg);

void mtf_exit(int status);
int  mtf_error(char *string, int code);

/* low-level locks */

int mtf_lock_init(mtf_lock lock);
int mtf_lock_free(mtf_lock lock);
int mtf_lock_acquire(mtf_lock lock);
int mtf_lock_release(mtf_lock lock);

/* core TCB functions */

int mtf_tcb_init(mutex_tcb *tcb, mtf_tcb_func *func, void *arg);
int mtf_tcb_start(mutex_tcb *tcb);
int mtf_tcb_yield(void);
void mtf_tcb_exit(void);
mutex_tcb *mtf_tcb_current(void);

/* malloc and free */

void *mtf_malloc(size_t size);
void  mtf_free(void *ptr);

/* custom primitives */

int mutex_block (mutex_container *b_container, mtf_lock *lock);
int mutex_unblock (mutex_container *b_container);
```

Figure 7.6: Mezcla Primitives

```
int mutex_create(mtf_tcb_func *func, void *arg)
{
        mutex_tcb *new;

        new = mutex_tcb_alloc();
        new->stacksize = mtf_stacksize;

        mtf_lock_init(new->lock);

        mtf_tcb_init(new, (mtf_tcb_func *) func, arg);

        /* set global thread count */
        mtf_lock_acquire(mutex_tcount_lock);
        mutex_tcount++;
        mtf_lock_release(mutex_tcount_lock);

        mtf_tcb_start(new);

        return 0;
}
```

Figure 7.7: MutexThreads: `mutex_create()`

```
int mutex_lock(mutex *m)
{
        mtf_lock_acquire(m->lock);

        if ((m->value) == 0) {
                m->value = 1;
        } else {
                mutex_block(&(m->container), &(m->lock));
        }

        mtf_lock_release(m->lock);
}
```

Figure 7.8: MutexThreads: `mutex_lock()`

then the mutex is not available and the calling thread must block. Blocking is achieved by calling the custom primitive `mutex_block()` and passing it a container and a lock. All calls to the custom block and unblock primitives must obey certain locking conventions (explained in Section 8.1.2).

```
int mutex_unlock(mutex *m)
{
        mtf_lock_acquire(m->lock);

        if (mutex_tcb_head(&(m->container)) != ((mutex_tcb *) NULL)) {
                mutex_unblock(&(m->container));
        } else {
                m->value = 0;
        }

        mtf_lock_release(m->lock);
}
```

Figure 7.9: MutexThreads: `mutex_unlock()`

Figure 7.9 lists the code for `mutex_unlock()`. This function is the complement of `mutex_lock()`. If another thread is waiting for the mutex, it uses the custom primitive `mutex_unblock()` to unblock a waiting thread. Otherwise, it sets the mutex value to 0.

7.2.5 Thread Generation

So far, we have seen the MutexThreads client description, the `tgen` generated Mezcla primitives, and the implementation of some of the MutexThreads interface functions. All that remains is the generated *implementation* of the Mezcla primitives for a particular target (the result of Step 5 from Section 7.1). The `tgen` program requires a client description file and a target selection (by target we mean *target type*). Currently, the Mezcla prototype supports five target types: direct switch (**ds**), preswitch (**ps**), preswitch preemption (**pspr**), preswitch multiprocessor (**psmp**), and pthreads (**pt**). Most of these target types are implemented on four `target platforms`: Linux/x86, IRIX/MIPS, Solaris/Sparc, and OSF1/Alpha. The implementations for these targets are complex; for our example, we will look only at the implementation of `mutex_block()` for two targets types: **ds** and **psmp**. See Section 8.3 for a detailed presentation of target descriptions and the supported target types. See Section 8.4 for description of the algorithms used by `tgen` for generating thread systems.

```
int mutex_block (mutex_container *b_container, mtf_lock *lock)
{
        mutex_tcb *b_tcb, *s_tcb;

        /* get next tcb to schedule */
        s_tcb = mutex_tcb_get(mtf_scb_field(ready_container));
        b_tcb = mtf_tcb_ptr;

        if (s_tcb == NULL) {
                mtf_exit(0);
        }

        mutex_tcb_put(b_tcb, b_container);

        mtf_tcb_set(s_tcb);

        /* switch to new tcb */
        ds_chg_context(s_tcb->sp, b_tcb->sp);
}
```

Figure 7.10: MutexThreads: Direct Switch `mutex_block()`

The **ds** target is a non-preemptive, uniprocessor target that uses a direct context switch method. Figure 7.10 lists the generated code for the **ds** version of `mutex_block()` and shows five main characteristic features. First, most of the names are client-specific, e.g., `mutex_block`, `mutex_container`, and `mutex_tcb`. Second, moving TCBs to and from containers is accomplished with the client-specified scheduling operations, e.g., `mutex_tcb_get()` and `mutex_tcb_put()`. Third, the macro `mtf_tcb_ptr` is used to determine the current thread and `mtf_tcb_set()` is used to set the current thread. Fourth, the generated code for `mutex_block` includes the variables `b_tcb` and `b_container` in the scope of the client-specified block code, `mutex_tcb_put()`. (Recall from Section 7.2.3 that these variables are used for defining the custom synchronization primitives in the client description file.) Fifth, a context switch is achieved with the function `ds_chg_context()`, which is provided by a Mezcla support library for context switching.

The **psmp** target is a non-preemptive, multiprocessor target that uses the

```
int mutex_block_helper(qt_t *sp, void *b_tcb,
                       mutex_container *b_container)
{
        ((mutex_tcb *) b_tcb)->sp = sp;

        mutex_tcb_put(b_tcb, b_container);
        mtf_lock_release(*(((mutex_tcb *) b_tcb)->prev_lock));
}

int mutex_block (mutex_container *b_container, mtf_lock *lock)
{
        mutex_tcb *b_tcb, *s_tcb;

        /* get next tcb to schedule */
        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
        s_tcb = mutex_tcb_get(mtf_scb_field(ready_container));
        mtf_lock_release(mtf_scb_field(ready_container_lock));

        b_tcb = mtf_tcb_ptr;

        /* get idle tcb if no more ready tcbs */
        if (s_tcb == NULL) {
                s_tcb = mtf_ecb_field(idle);
        }

        mtf_tcb_set(s_tcb);

        /* set the old tcb previous container lock */
        b_tcb->prev_lock = lock;

        /* switch to new tcb */
        QT_BLOCK((qt_helper_t *) mutex_block_helper, b_tcb,
                 (void *) b_container, s_tcb->sp);

        mtf_lock_acquire(*(b_tcb->prev_lock));
}
```

Figure 7.11: MutexThreads: Preswitch Multiprocessor `mutex_block()`

preswitch method for context switching. Figure 7.10 lists the generated code for the **psmp** version of `mutex_block()`. Similar to the **ds** version of `mutex_block()`, the **psmp** version uses client-specific names for data types and operations. However, unlike the **ps** version, the **psmp** version has a significantly different implementation. Because **psmp** provides true parallelism, locks must be used to ensure mutual exclusion to shared data structures, such as the global ready container. Also, unlike **ds**, if no ready thread is available, then an *idle* thread must be used. Finally, the **psmp** target uses QuickThreads for implementing the preswitch method. See Section 8.5.1.2 for a discussion of the QuickThreads interface.

7.3 Summary

This chapter presented an overview of how to use the Mezcla prototype and an example of a simple Mezcla-based thread system called MutexThreads. The example shows how the Mezcla prototype separates client concerns from target concerns. Based on this separation, a client does not have to worry about target-specific issues (such as context switching methods). This example also shows how the client can specify particular naming conventions, scheduling operations, and custom synchronization primitives that are integrated into a target during thread generation. Chapter 8 will present the complete details of the prototype implementation.

Chapter 8

The Mezcla Thread Framework: Prototype Implementation

This chapter describes a prototype implementation of the Mezcla thread framework that is based on the design issues presented in Chapter 6. We present the prototype in terms of its major components: the primitives, the client description format, the target description format, the thread generation tool, and support libraries. Figure 8.1 gives a more detailed version of Figure 6.1 in terms of the framework components; it also provides a visual roadmap to this chapter.

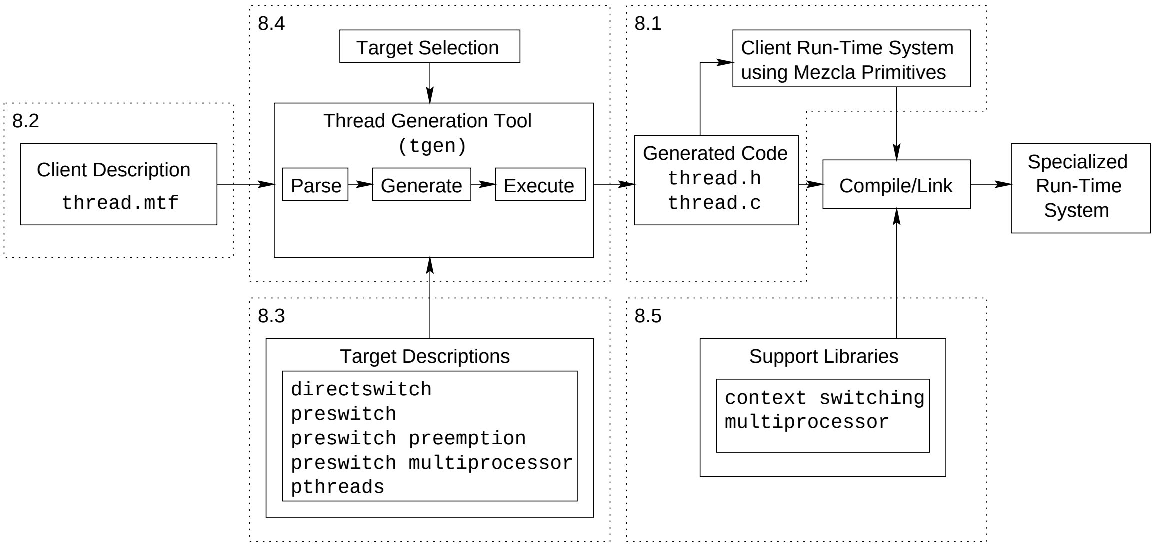

Figure 8.1: The Mezcla Thread Framework Prototype

The prototype implementation is a hybrid of the template and compiler approaches described in Section 6.5.4. The primary goal of this implementation is to demonstrate the feasibility of the Mezcla thread framework. Some of the material presented in this chapter repeats parts of Chapter 7 for completeness. The prototype described in this chapter is used for the thread system implementations and experiments presented in Chapter 9.

The most challenging components to develop were the client and target specifications. The main elements of the client specification format include specialized TCBs, TCB containers, scheduling, and synchronization primitives. Client specifications are very simple to learn and easy to use; they resemble Yacc grammars in that Mezcla rules are intermixed with C code. Target specifications are more complicated because it is difficult to capture all of the relevant details of a particular target platform.

To manage the complexity of target descriptions, we use a technique called *template-driven code generation* [102]. The Mezcla prototype is heavily based on the example code from [102]. The basic idea is to write targets as annotated C code that serve as executable templates (the annotations are written in Perl [115]) and make it easy to access the abstract syntax tree for a parsed client description and to create new C code based on the client information. The thread generation tool (called `tgen`) parses a client description, creates an executable template, and then executes the template to generate a specialized thread system.

Finally, the prototype relies on some core libraries for context switching and multiprocessor primitives.

8.1 Primitives

To develop a Mezcla thread system, a programmer must write the *client-half* of the thread system using Mezcla primitives. The Mezcla prototype consists of two types of primitives: *fixed* and *custom*. The fixed primitives exist for every Mezcla client, whereas the custom primitives are defined by the client. Both fixed and custom primitives are *specialized* for a particular client/target pair during thread generation. Even some of the fixed primitives are modified to reflect naming conventions determined by the client.

```
/* initialization and system functions */

int  mtf_init(mtf_tcb_alloc_func *tcb_alloc,
              mtf_tcb_free_func *tcb_free,
              mtf_stack_alloc_func *stack_alloc,
              mtf_stack_free_func *stack_free);

int  mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func,
               void *main_arg);

void mtf_exit(int status);
int  mtf_error(char *string, int code);

/* low-level locks */

int mtf_lock_init(mtf_lock lock);
int mtf_lock_free(mtf_lock lock);
int mtf_lock_acquire(mtf_lock lock);
int mtf_lock_release(mtf_lock lock);

/* core TCB functions */

int mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg);
int mtf_tcb_start($mtf_tcb *tcb);
int mtf_tcb_yield(void);
void mtf_tcb_exit(void);
$mtf_tcb *mtf_tcb_current(void);

/* malloc and free */

void *mtf_malloc(size_t size);
void  mtf_free(void *ptr);
```

Figure 8.2: Fixed Primitives

8.1.1 Fixed Primitives

The fixed thread primitives are listed in Figure 8.2. Note that most of the primitives have integer return values. In principle, the primitives should report success or failure; in practice, though, many do not return meaningful values. This situation should be rectified in future versions of the prototype.

The initialization and system functions are used to initialize, start, and exit multi-threaded execution. The `mtf_init()` function simply gives the thread system pointers to client functions for TCB and stack allocation, which are set in the thread system's SCB (system control block). In addition to initializing these callback pointers, `mtf_init()` performs internal and target-specific initialization, e.g., global container initialization, ECB (execution control block) allocation, and multiprocessor setup. Multi-threaded execution does not begin until `mtf_start()` is called. This function requires a pointer to an initialized TCB — by convention called the *main* TCB — that will be the first thread executed when multi-threaded execution begins. Generally, `mtf_start()` will not return; thread systems usually exit explicitly using `mtf_exit()`. Notice that the type of `*main_tcb` is `$mtf_tcb` (and that other client-defined types are also prefixed with a "$"). This type is a placeholder for the TCB type provided by the client; it will be substituted with a real type during thread generation.

The `mtf_error()` function is used to print an error message and then exit the program, while `mtf_exit()` simply exits the application. It is important for a Mezcla client to call `mtf_error()` or `mtf_exit()` to exit a program because some of the targets have special exiting requirements. For example, some multiprocessor targets require kernel threads to be killed explicitly; otherwise, they will be left to run, consuming memory and processor cycles.

The low-level lock primitives are used to provide mutual exclusion to shared data in the presence of multiprocessors, preemptive scheduling, or both. The lock data type is `mtf_lock`, which is mapped to the appropriate target-supplied lock data type. The `mtf_lock_init()` function initializes a lock and the `mtf_lock_free()` function deallocates a lock. The `mtf_lock_acquire()` function acquires a lock. If the lock is available, then `mtf_lock_acquire()` will simply return; otherwise, it will block the calling thread until the lock is available. The `mtf_lock_release()` function resets a lock, allowing it to be acquired again, and possibly unblocks a thread waiting for the lock.

Currently, all Mezcla clients must implement threads assuming true parallelism. This conservative approach handles all types of targets; for targets without true parallelism or preemption, the lock primitives are ignored (they are macros that map to empty strings). For multiprocessor targets, they map to spinlocks or operating system supplied, multiprocessor-safe locks. For preemption targets, the locks prevent context switching in a critical section. Finally, for the Pthreads target, the low-level locks map to Pthreads mutex locks.

The core TCB functions are essentially used to start and stop threads. The `mtf_tcb_init()` function is used to initialize a TCB. Because the data fields in a Mezcla TCB are shared between the client and the target, it is expected that the client allocates memory for the new TCB and initializes any client-specific TCB fields. The client is also expected to set the TCB field `mtf_stacksize`. A global variable, `mtf_stacksize`, is available to provide a target-specific default stack size. The `mtf_tcb_init()` function is passed a `mtf_tcb_func` pointer to establish the start function for the new thread. The `arg` pointer is used to pass an initial value to the start function. A thread will not begin executing until the client calls `mtf_tcb_start()` with a pointer to an initialized TCB. Generally, calling `mtf_tcb_start()` results in putting the TCB on the ready queue.

The `mtf_tcb_yield()` function is used to allow other threads to be scheduled. If another thread is not ready, `mtf_tcb_yield()` simply returns and the calling thread continues executing. Otherwise, the current thread is put into a global ready container and a ready thread is allowed to run. Threads must exit with `mtf_tcb_exit()` to execute target-specific exit code. Generally, if a client requires any special exit functionality, the client will export a wrapper function that provides custom exit code and eventually calls `mtf_tcb_exit()`.

The `mtf_tcb_current()` function is used to return a pointer to the TCB of the currently executing thread. This function, which is often implemented as a macro, returns a pointer of type `$mtf_tcb`. This function can be used to implement more sophisticated forms of thread-specific data or to directly access client-specific TCB fields for implementing language run-time systems.

Finally, Mezcla clients must use `mtf_malloc()` and `mtf_free()` to allocate and deallocate memory, respectively, because not all operating system supplied versions of `malloc()` and `free()` are thread safe. The `mtf_malloc()` and `mtf_free()` functions are macros that are replaced with target-specific versions during compilation.

8.1.2 Custom Primitives

Currently, the prototype only supports two types of custom primitives: thread *blocking* and thread *unblocking*. These two operations are fundamental to most high-level synchronization mechanisms found in thread libraries and concurrent programming languages. While it is possible to build a Mezcla thread system without custom blocking primitives, such a thread system would be very limited with respect to synchronization. This section describes how a client uses custom primitives. Section 8.2 shows how a client specifies custom primitives, Section 8.3 shows how targets provide support for custom primitives, and Section 8.4 shows how the thread generation tool, `tgen`, creates the primitives.

Figure 8.3 lists the templates for the two types of custom primitives supported in the prototype. Block and unblock primitives are grouped in pairs. That is, each block primitive will always have a corresponding unblock primitive and vice versa. A client can create multiple block and unblock primitives for a single thread system, which is useful for handling multiple synchronization mechanisms in the same library or run-time system.

```
int $block_name ($cname *b_container, mtf_lock *lock);
int $unblock_name ($cname *b_container);
```

Figure 8.3: Custom Primitives

The items `$block_name`, `$unblock_name`, and `$cname` are determined by the client in the client description file. The `$block_name()` primitive is used to block (queue) the current thread into `b_container` and schedule another thread if one is available. A client-supplied container *put* function is used to place the current TCB into `b_container`. The caller must acquire `lock` with `mtf_lock_acquire()` before invoking `$block_name()`. The `$block_name()` function will release the `lock` before running a new thread; it will also return with the `lock` acquired (which the caller must eventually release). Figure 8.4 shows skeleton code for how a client should use `$block_name()`.

The `$unblock_name()` primitive is used to unblock a currently blocked thread. A client supplied container *get* function is used to remove a TCB from `b_container`. Similar to `$block_name()`, the caller of `$unblock_name()` must acquire the lock associated with `b_container`. Unlike `$block_name()`, `$unblock_name()` does not manipulate the `lock`, so it is not passed as an argument. Figure 8.5 shows skeleton code for how a client should use

```
int client_block(client_object *co)
{
        mtf_lock_acquire(co->lock);
        ...
        $block_name(&(co->container), &(co->lock));
        ...
        mtf_lock_release(co->lock);
}
```

Figure 8.4: A Template for Using the Custom Block Primitive

`$unblock_name()`.

```
int client_unblock(client_object *co)
{
        mtf_lock_acquire(co->lock);
        ...
        $unblock_name(&(co->container));
        ...
        mtf_lock_release(co->lock);
}
```

Figure 8.5: A Template for Using the Custom Unlock Primitive

Recall from Section 6.1.1 that the reason for acquiring and holding locks across calls to the Mezcla primitives is to avoid redundancies found in the traditional coupling of run-time systems and thread systems. In the unblock example, `co->lock` is only acquired once for the primitive. In the block example, `co->lock` is acquired twice at most: once before blocking and once when the thread resumes execution if the particular target resumes thread execution with `co->lock` unlocked. Currently, Mezcla does not optimize the case where the target resumes a thread with `co->lock` released and the client immediately releases `co->lock`. Using `$block_name()` and `$unblock_name()` in this fashion allows the actual blocking or unblocking primitives to be specialized on a per-target basis.

To better understand how to use the custom primitives, we look at a pair of block and unblock primitives for the SemThreads library. SemThreads is a simple thread system based on semaphores (described in detail in Section 9.1). Figure 8.6 shows a pair of custom primitives generated for the SemThreads library. In this example, `$block_name` =

`sem_block`, `$unblock_name` = `sem_unblock`, and `$cname` = `sem_container`.

```
int sem_block (sem_container *b_container, mtf_lock *lock);
int sem_unblock (sem_container *b_container);
```

Figure 8.6: Custom SemThreads Primitives for Semaphore Operations

Figure 8.7 lists the SemThreads code for the `sem_P()` operation. Notice that most of the function and variable names are client defined. In this example, the client code uses two *fixed* primitives, `mtf_lock_acquire()` and `mtf_lock_release()`, and one *custom* primitive, `sem_block()`. The algorithm in `sem_P()` is a straightforward implementation of the **P** operation (see Section 2.1.2).

```
int sem_P(sem *s)
{
        mtf_lock_acquire(s->lock);
        if ((s->value) > 0) {
                (s->value)--;
        } else {
                (s->blocked)++;
                sem_block(&(s->container), &(s->lock));
        }
        mtf_lock_release(s->lock);
}
```

Figure 8.7: SemThreads Implementation of the **P** Operation

For completeness, Figure 8.8 lists the SemThreads code for the `sem_V()` operation. The full SemThreads implementation, given in Appendix C, also uses another pair of custom block and unblock primitives to implement fork/join synchronization.

8.2 Client Descriptions

The client description format enables a programmer to easily specify everything about a thread system that is client specific. The format is a collection of statements, where the type of statement is determined by the first token on a non-empty line. Each statement is parsed differently, depending on the statement type. However, compound

```
int sem_V(sem *s)
{
        mtf_lock_acquire(s->lock);
        if ((s->blocked) > 0) {
                (s->blocked)--;
                sem_unblock(&(s->container));
        } else {
                (s->value)++;
        }
        mtf_lock_release(s->lock);
}
```

Figure 8.8: SemThreads Implementation of the **V** Operation

statements generally look like C blocks using braces to group code or data. The `#` character is used for comments; every thing after a `#` character up to the next newline is ignored. While statements are not required to appear in any fixed order, it is useful to divide client descriptions into three parts: the preamble, data statements, and code statements. The preamble is used for basic configuration statements, data statements establish client-specific data structures, and code statements allow a client to specify scheduling actions and to define custom primitives. Figure 8.9 lists the basic format for a Mezcla client description.

8.2.1 Preamble

Currently, the preamble consists of a collection of `set` statements. A `set` statement simply associates (sets) a variable name to a value. The syntax for the `set` statement is:

```
set <variable> <value>
```

Neither `variable` nor `value` can contain spaces. Different `set` variables have different meanings. Two `set` variables are currently used: `name` and `client_inc`.

The `name` variable is used to determine the prefix for the file names of generated code. For example, if `name` = foo, then `tgen` will output generated code to the files `foo.h` and `foo.c`. The `client_inc` variable is used to provide a client-specific include file to the target templates. It is used to give the target templates access to client-specific code or definitions used in the data or code statements.

```
# Mezcla Client Description

# Preamble

set name <prefix>
set client_inc <filename>

# Data

data tcb <tcb_name> {
        <client tcb fields>
        mtf_lock lock;
}

data tcb_container <container_name> {
        <client tcb container fields>
}

# Code

code sched <container_name> {
        set init <client_init>
        set put  <client_put>
        set get  <client_get>
        set head <client_head>
}

codel sync <type> {
        block <block_name> {
                <client blocking code>
        }
        unblock <unblock_name> {
                <client unblocking code>
        }
}
```

Figure 8.9: The Client Description Format

8.2.2 Data Description

Data description statements are used to define the client-specific fields of data structures shared between the client and the thread system. The syntax for the `data` statement is:

```
data <targetname> <clientname> {
        C data declarations
}
```

The `targetname` is the name of the Mezcla data structure, while `clientname` is what the client wants to call the `targetname` data structure. The Mezcla generated code will include a C typedef for the new, shared data structure with the name `clientname`. Currently, two shared data structures are supported: the TCB (`targetname` = `tcb`) and TCB containers (`targetname` = `tcb_container`).

8.2.2.1 Thread Control Blocks

The `tcb` is the central shared data structure for Mezcla specialized thread systems (see Section 6.2.1.2 for a discussion of TCBs). Figure 8.10 shows the TCB description for SemThreads. The SemThreads TCB fields include an exit status value, `exited`; a join TCB, `join_tcb`; a join value, `join_value`; and a link for putting TCBs into containers, `next`. (The SemThreads client uses `sem_tcb` as the type name for the TCB data type.) Currently, the client must by convention include a `lock` field. In addition, all target TCBs have an `mtf_stacksize` field. Before calling `mtf_tcb_start()`, the client is responsible for setting `mtf_stacksize` and initializing `lock`.

```
data tcb sem_tcb {
        int exited;
        struct sem_tcb *join_tcb;
        void *join_value;
        struct sem_tcb *next;
        mtf_lock lock;
}
```

Figure 8.10: SemThreads TCB Description

8.2.2.2 Thread Control Block Containers

The tcb_container is used by the client and the target templates to store TCBs (see Section 6.2.1.3). For example, the target templates presented in Section 8.3.3 use the tcb_container data structure as a ready container to hold threads that are not blocked and are eligible to run. The TCB container information is also needed by the targets to generate specialized data structures (such as the ECB and SCB) and specialized code (such as blocking and unblocking primitives).

```
data tcb_container sem_container {
        struct sem_tcb *head;
        struct sem_tcb *tail;
}
```

Figure 8.11: SemThreads TCB Container Description

Figure 8.11 lists the SemThreads TCB container description. It is the responsibility of the client code to make sure TCBs and TCB containers are compatible with each other. This gives the client complete flexibility in defining the relationship between TCBs and TCB containers.

8.2.3 Code Description

Code description statements are used to establish client-specific code for scheduling and custom synchronization primitives. The Mezcla prototype supports two types of code description statements: `code` and `codel`. The `code` statements are simply used to associate a code `class` with a particular data type (as defined with a `data` statement) and a set of predefined operations specific to the `class`. The syntax for the `code` statement is:

```
code <class> <type> {
       <set statements>
}
```

The `class` field must be supported by the Mezcla prototype. Currently, only the `sched` class is supported (see Section 8.2.3.1 below). The body of a `code` statement consists of a series of `set` statements (see Section 8.2.1).

The `codel` (short for "code list") statement is used to specify code for custom primitives in a particular `codel class`. The syntax for the `codel` statement is:

```
codel <class> <type> {
        operation_1 <clientname> {
                <client C code>
        }
        operation_2 <clientname> {
                <client C code>
        }
        ...
        operation_N <clientname> {
                <client C code>
        }
}
```

The `type` and `operation` names are determined by a particular class. In addition, each `codel` class dictates the *execution environment* in which the client C code will execute. Currently, the prototype supports only one `codel` class, the `sync` class (see Section 8.2.3.2).

8.2.3.1 Scheduling

The target scheduler is specialized by using a `code` statement with the `sched` class. The syntax for `sched code` is:

```
code sched <container_type> {
       set init <client_init>
       set put  <client_put>
       set get  <client_get>
       set head <client_head>
}
```

The `sched` class requires a `container_type`, defined with a `data` statement. The `sched` class also requires four operations to be defined: `init`, `put`, `get`, and `head`. The operations are used by the target templates to initialize a ready container and to perform basic container operations. The `sched` operations have the following implied types:

```
int client_init(container_type *container);
int client_put(client_tcb *tcb, container_type *container);
client_tcb *client_get(container_type *container);
client_tcb *client_head(container_type *container);
```

Because the client specifies the TCB, the TCB container, and the scheduling operations, the client can dictate how threads are scheduled. For SemThreads, a TCB container is a simple linked list of TCBs and the `sched` operations are simple FIFO queue operations. However, the TCB container could be defined as a more sophisticated data structure and the operations could be more complex.

8.2.3.2 Synchronization

Custom synchronization primitives are specified using `codel` statements with the `sync` class. A single `codel sync` statement defines a pair of block and unblock primitives. A client description file can have multiple `codel sync` statements, so multiple custom block/unblock primitives can be generated. The format is:

```
codel sync <type> {
        block <block_name> {
                <client C code>
        }
        unblock <unblock_name> {
                <client C code>
        }
}
```

The `type` field specifies the type name for the *destination* variable of the `block` operation and the *source* variable of the `unblock` operation. The client code for both `block` and `unblock` assumes that it will have access to two `execution environment` variables, `b_tcb` and `b_container`. For the `block` operation, the client must supply code that will place `b_tcb` (which is the blocking TCB) into `b_container` (which is the blocking container passed in by the client code). Similarly, for the `unblock` operation, the client must supply code that will remove a TCB from `b_container` (which is the unblock container passed in by the client code) and set `b_tcb` to point to this TCB. This separation of blocking function from blocking mechanism allows a target template to place the blocking and unblocking operations at the appropriate place in the generated code for a particular target, while allowing the client to specify the manner in which TCBs are blocked or unblocked.

```
codel sync sem_container {
        block sem_block {
                sem_tcb_put(b_tcb, b_container);
        }
        unblock sem_unblock {
                b_tcb = sem_tcb_get(b_container);
        }
}
```

Figure 8.12: SemThreads `codel` Statement for **P** and **V** Synchronization

Figure 8.12 lists the SemThreads specification for the `sem_block()` and `sem_unblock()` primitives from Figure 8.6. In this case, the client code blocks are just simple calls to the SemThreads container functions. However, in general, the code blocks can be more complex than a single C statement. Sections 8.3 and 8.4 describe how the primitives in Figure 8.6 are generated from the specification in Figure 8.12.

8.3 Target Descriptions

In the Mezcla prototype, target descriptions are implemented as templates. A target template is a skeleton thread system written in C with Perl annotations, which are used to fill in the templates with information extracted from the client description file. One can think of the target template format as an advanced macro system. The Perl code enables the template to walk the abstract syntax tree generated from the client specification described in Section 8.2. The thread generation tool, `tgen`, passes an abstract syntax tree to a target template to create the *Mezcla half* of the thread system.

It is important to understand the difference between a *target type* and a *target platform*. A target type is a general approach to implementing a thread system, whereas a target platform is a specific operating system and processor architecture. A single target type may be supported by several different target platforms. For example, the preswitch multiprocessor (**psmp**) target type implements non-preemptive, multiprocessor enabled thread primitives and is supported on Linux/x86, IRIX/MIPS, and Sparc/Solaris. Not all

Annotation	Use
`@//`	Start a template comment
`@openfile`	Open a file for output (used to open generated files)
`@perl`	Execute arbitrary Perl code
`@visit`	Visit a parse node in the abstract syntax tree and make parse node variables visible
`@vend`	End a `@visit` and return to the parent parse node
`@foreach`	Iterate over a list of parse nodes (visit each node in turn)
`@end`	End a `@foreach`

Table 8.1: Target Template Annotations

target types are supported by all target platforms. For example, the **psmp** target type is not supported on OSF1/Alpha because this platform does not provide operating system support for directly creating kernel-level threads.

The separation of target type from target platform makes it somewhat easier to port Mezcla to new platforms. Currently, only the support libraries described in Section 8.5 need to be written for a new target platform. Once the support libraries are written, the target types should work for the new platform. It is easiest to write a new target description by starting from one of the available descriptions.

The following sections describe the components of Mezcla target descriptions: template annotations, a template format, and the currently supported target types. Appendix A lists the source code for all the supported target templates. The reader might want to browse these templates while reading the following sections.

8.3.1 Template Annotations

As mentioned above, a target template is simply a skeleton thread system written in C and annotated with Perl code. Template files typically have a `.tpl` extension. The Perl annotations allow a template to access client-specific information in the form of an abstract syntax tree. The C code is written as one would normally write C code with the exception that Perl-style variables (e.g., `$varname`) will be substituted during code generation. Each Perl annotation begins with the `@` symbol. Table 8.1 lists the supported annotations.

The thread generation tool, `tgen`, reads a target template file and transforms it into an executable Perl program. Template lines that are not annotations are passed without any variable substitutions to the current output file. Template annotations are translated into Perl code.

Template comments begin with `@//`. Everything after `@//` up to the next newline is ignored. Template comments will not appear in the generated code; they are used to comment the template itself. On the other hand, regular C-style comments will appear in the generated code.

The `@openfile` annotation is used to open an output source file. All output from the template will be sent to the current output file. Calling `@openfile` will also close a previously opened file. Mezcla target templates typically use `@openfile` twice: once to open a `.h` file and once to open a `.c` file. The syntax for `@openfile` is:

```
@openfile <filename>
```

Typically, the `filename` will be based on a client specified name, for example:

```
@openfile ${name}.h
```

The `@perl` annotation allows the template to execute arbitrary Perl code. This is useful for setting global variables that can be used in subsequent variable substitutions and for performing code generation that cannot be achieved with simple variable substitution, such as generating output from a list. The syntax for `@perl` is:

```
@perl <perl code>
```

The `@perl` annotation is typically used to set variables, for example:

```
@perl @perl $varname = $setname;
```

It is also commonly used to generate a code fragment from a list of strings, for example:

```
@perl for (@$list) { output("\t$_\n"); }
```

The `(@$list)` string is Perl syntax for accessing all the elements of an array. The `$_` symbol

Node Name	Node Type	Symbol	Symbol Type
`root`	*root*	`name` `client_inc`	string string
`tcb`	*data*	`name` `list`	string list of strings
`tcb_container`	*data*	`name` `list`	string list of strings

`sched`	*code*	`cname`
		`init`
		`get`
		`put`
		`head`
			string
			string
			string
			string
			string

`sync`	*codel*	`cname`
		`block`
		`unblock`
			string
			data node
			data node

Table 8.2: Parse Node Types

is Perl syntax for accessing the current list element.

The `@visit` and `@vend` annotations are used to descend into and return from parse nodes, respectively. All of the current parse node variables are accessible for variable substitution in the C code and the Perl annotations. Initially, the template is executed at the *root* parse node. The root node contains the values of any top-level `set` variables as well as parse nodes for all other statements from the client description file.

The `@visit` annotation requires as a parameter a node name. Currently supported node names are: `tcb`, `tcb_container`, `sched`, and `sync`. Each node has a type, which defines what information will be available in a particular node. Table 8.2 lists the parse node names with their corresponding types and available symbols. The nodes correspond to statements from the client description format presented in Section 8.2.

Symbols whose types are node types must be visited with the `@visit` annotation. Currently, abstract syntax trees are very shallow, reflecting the shallow structure of the client description language. Not including the initial visit to the root node, recursive visits are required only on `codel` nodes.

Figure 8.13 demonstrates how to use `@visit` on a `data` node. This example shows how to generate a custom TCB from a `tcb data` node (this example comes from the **psmp** target template listed in Section A.2.4). The argument to `@visit` specifies the name of the node to visit. The template lines after the `@visit` annotation have access to all the symbols (variables) present in the node. For example, because `tcb` is a `data` node, the symbols `name` and `list` are made available to the client code.

```
/* thread control block */
@visit tcb
@perl $mtf_tcb = $name;
typedef struct $name {
        /* client supplied tcb fields */
        @perl for (@$list) { output("\t$_\n"); }

        /* target fields */
        void *stack;
        void *aligned_stack;
        qt_t *sp;
        mtf_lock *prev_clock;
} $name;
@vend
```

Figure 8.13: Using the `@visit` Annotation on `data` Nodes

The first `@perl` annotation in Figure 8.13 is used to set a global variable, `$mtf_tcb`, to the data type name for the TCB. The second `@perl` annotation is used to insert all of the client-supplied TCB fields into the generated TCB.

```
@visit sched
@perl $mtf_tcb_container = $cname;
@perl $mtf_tcb_container_init = $init;
@perl $mtf_tcb_put  = $put;
@perl $mtf_tcb_get  = $get;
@perl $mtf_tcb_head = $head;
@vend
```

Figure 8.14: Using the `@visit` Annotation on `code` Nodes

Figure 8.14 shows how to use `@visit` on a `sched` node. The `@perl` annotations are simple variable assignments.

The last two template annotations are `@foreach` and `@end`, which simply allow a template to "visit" all nodes with the same name. They are useful for iterating over several nodes that use the same code-fragment template. The `@foreach` annotation is used in Mezcla templates for `codel` nodes with the same name, which is useful for generating multiple custom synchronization primitives. Figure 8.15 shows how the **ds** target uses the `@foreach` annotation.

This section has given a synopsis of the different types of annotations that can be used in target templates. See Appendix A for complete listings of all the Mezcla target templates. Note that the template programmer must ensure that the resulting generated code provides all the primitives expected by the client and that these primitives behave as defined in Section 8.1.

8.3.2 Template Format

A proper target template must generate code to support all the primitives and global data described in Section 8.1. However, target templates are extremely *unconstrained*, as they are essentially C programs. To help the development of target templates, we established a primitive template format. While using this format is not necessary, it makes it easier to make uniform changes to all the templates. The basic format consists of the following sections (see Appendix A for the complete source to the supported target types):

- Global template variables used by both the `.c` file and the `.h` file.
- Generated header file (`name.h`)
  - Initial comments
  - Include files (necessary for the resulting thread systems)
  - Macros (control block access macros and target-specific macros)
  - Data type definitions (locks, TCB, ECB, SCB)
  - Global data definitions (`mtf_stacksize` and control block pointers)
- Generated C file (`name.c`)
  - Initial comments
  - Include files (`name.h`)
  - Global data

```
@foreach sync
@visit block
/* block current tcb and schedule a new tcb */
int $name ($cname *b_container, mtf_lock *lock)
{
        $mtf_tcb *b_tcb, *s_tcb;

        /* get next tcb to schedule */
        s_tcb = $mtf_tcb_get(mtf_scb_field(ready_container));
        b_tcb = mtf_tcb_ptr;

        if (s_tcb == NULL) {
                /* no tcbs to schedule */
                mtf_exit(1);
        }

        @perl for (@$list) { output("\t$_\n"); }

        mtf_tcb_set(s_tcb);

        /* switch to new tcb */
        ds_chg_context(s_tcb->sp, b_tcb->sp);
}
@vend

@visit unblock

/* unblock a tcb */
int $name ($cname *b_container)
{
        $mtf_tcb *b_tcb;
        /* get tcb from container */
        @perl for (@$list) { output("\t$_\n"); }

        /* put tcb on ready container */
        if (b_tcb != NULL) {
              $mtf_tcb_put(b_tcb, mtf_scb_field(ready_container));
        }
}
@vend
@end
```

Figure 8.15: Using the `@foreach` Annotation on `codel` Nodes

Target	Full Name	Filename	Locking	Thread ID	Exit	Idle
ds	Direct Switch	`mtf_ds.tpl`	no	direct	yes	no
ps	Preswitch	`mtf_ps.tpl`	no	direct	no	no
pspr	Preswitch Preemption	`mtf_pspr.tpl`	yes	direct	no	no
psmp	Preswitch Multiprocessor	`mtf_psmp.tpl`	yes	indirect	no	yes
pt	Pthreads	`mtf_ds.tpl`	yes	indirect	no	no

Table 8.3: Target Template Types

- Internal functions
- Fixed primitives
- Custom primitives

8.3.3 Target Types

We now look at the target types currently implemented in the Mezcla prototype. For each target type, we consider the following issues: initialization considerations, locking, control block organization, thread identification, thread scheduling, thread exiting, and the custom synchronization primitives. Table 8.3 gives a brief summary of each target type. The Locking column indicates whether locking is needed. The Thread ID column indicates whether TCB identification is direct (through a pointer) or indirect (through a register or function). The Exit column indicates whether the target uses an exit thread. Finally, the Idle column indicates whether idle threads are used. The complete source for all the target types is listed in Appendix A.

8.3.3.1 Direct Switch (ds)

The Direct Switch target, **ds**, is based on the direct switch assembly code presented in Section 8.5.1. The source code for **ds** is listed in Section A.2.1. This target type implements a non-preemptive, uniprocessor thread system. The target-supplied TCB fields simply include a stack pointer and a context pointer used by the direct switch code. The initialization code for **ds** is straightforward: it simply initializes the SCB (System Control Block) and allocates and initializes a single ECB (Execution Control Block)[1]. In addition,

[1]Recall from Section 6.2.1 that each Mezcla thread system has a single SCB and one or more ECBs depending on the target type.

the initialization code must allocate a TCB and a stack for the exit thread (see below).

Because only one thread can be executing at any given time and threads cannot be preempted, no low-level locking is required. Therefore, the `mtf_lock_*` primitives are null macros. Thread identification is accomplished by directly accessing the current thread field in the ECB.

Using the direct switch method for context switching requires that a special exit thread is used when a thread exits. It is not possible to free the exiting thread's stack while it is running. Therefore, the exiting thread switches to an exit thread so that it can safely deallocate the exiting thread's stack and TCB.

The strategy used by the code for a custom `block` primitive is to retrieve a TCB from the SCB's ready container. If a TCB is available, the current TCB field in the ECB is set to the new TCB; if not, the program terminates. Next, client-specific code for blocking the current TCB is executed. The client code has access to both `b_tcb` (the current thread) and to `b_container` (the blocking container). The current TCB field in the ECB is set to the new thread and a direct context switch is used to resume execution of the new thread. The custom `unblock` primitive simply uses the client-specific code to retrieve a TCB from a given block container and puts it into the ready container.

8.3.3.2 Preswitch (ps)

The preswitch target, **ps**, is based on the QuickThreads preswitch code presented in Section 8.5.1 (the source code for **ps** is listed in Section A.2.2). This target is similar to the **ds** target with a few exceptions. Like the **ds** target, the **ps** target implements a non-preemptive, uniprocessor thread system. The target supplied TCB fields include a stack pointer, an aligned stack pointer, and a current stack pointer, which is used by QuickThreads as a context handle. The initialization code for **ps** is similar to **ds**, except **ps** does not need to allocate an exit thread. Also similar to **ds**, the **ps** target does not require low-level locking and thread identification is accomplished by directly accessing a current thread field in the ECB.

The exit code for the **ps** target does not require an extra exit thread because a preswitch context switch allows any deallocations to execute inside a helper function on the stack of the next chosen thread.

The strategy used by the code for a custom `block` primitive is a bit more complicated than in the **ds** target due to the way preswitch works. The **ps** `block` primitive starts the same as the **ds** primitive by retrieving a TCB from the SCB's ready container. Also similar to **ds**, if a TCB is available, then the current TCB field in the ECB is set to the new TCB; otherwise, the program terminates.

However, the next step differs from the **ds** strategy in that preswitch is used to switch to the stack of the new TCB. Before restoring the context of the new TCB, a helper function is executed using the new TCB's stack. The helper function is passed a pointer to the old TCB and a pointer to a block container. Before resuming the new thread, the helper function saves the context of the old thread in the old thread's TCB and then executes the client-specific blocking code. As in the **ds** strategy, the **ps** strategy ensures that the client code has access to **b_tcb** and **b_container**.

8.3.3.3 Preswitch Preemption (pspr)

The preswitch preemption target, **pspr** (listed in Section A.2.3), is identical to the preswitch target except that a timer signal is used to generate an interrupt and locking is need to protect critical sections. The `mtf_lock_*` functions are designed to "disable" signals. The `sigpending()` system call could be used to mask out the timer signal. However, `sigpending()` must enter the kernel, so it incurs the cost of crossing the user/kernel boundary. To avoid this cost, we use two global variables, `preempt_mask` and `preempt_state`, that effectively delay the delivery of a timer signal. The `preempt_mask` variable is used to disable timer signals. Each time `mtf_lock_acquire()` is called, `preempt_mask` is incremented by 1. If a timer signal is delivered with `preempt_mask` > 0, then `preemp_state` is set to 1. Each time `mtf_lock_release()` is called, `preempt_mask` is decremented by 1. If `preempt_mask` $= 0$ and `preempt_state` $= 0$, then `mtf_tcb_yield()` is called to schedule a new thread. Note that the **pspr** version of the `mtf_lock_*` primitive ignores the lock argument. Acquire and release pairs are used simply to disable signals.

Unlike in **ds** and **ps**, all accesses to shared variables, such as the ready container and the block containers, must be protected by locks (to prevent race conditions). Thus, all modifications to the ready container are accessed inside a critical section protected by a target-supplied ready container lock, while the block containers are protected by client-supplied locks.

8.3.3.4 Preswitch Multiprocessor (psmp)

The preswitch multiprocessor target, **psmp**, uses the same basic strategy as **ps** and **pspr** for switching between threads, but it requires some significant additions to support multiple processors. This target depends heavily on the multiprocessor library, `libmp`, presented in Section 8.5.2. The source code for **psmp** is listed in Section A.2.4.

The largest difference between **psmp** and the other preswitch targets is the use of multiple ECBs and *idle* threads. The **psmp** target associates a new ECB with each virtual processor. Each ECB contains a pointer to the current thread for its virtual processor. The **libmp** virtual processor identification macro is used to determine the proper ECB for the current virtual processor.

With multiple processors, it is possible to have more processors than threads. When this happens, a virtual processor needs to switch to an idle thread where it can wait for a new thread to be created or to unblock. Therefore, an idle thread is allocated to each ECB (or virtual processor). Multiple idle threads are needed because different processors could be idle at the same time.

Finally, the **psmp** target maps the `mtf_lock_*` primitives to the `mtf_mp_lock_*` macros.

8.3.3.5 Pthreads (pt)

The Pthreads target, **pt**, maps the Mezcla primitives to the Pthreads library interface (see Section 2.4). The source code for **pt** is listed in Section A.2.5. The **pt** target is quite different from the other targets because Pthreads is a high-level interface.

A particular Pthreads implementation may support preemption, multiprocessor scheduling, or both. To protect shared data, the `mtf_lock_*` primitives are mapped to `pthread_mutex_*` functions. Thread identification is achieved with `pthread_getspecific()` and `pthread_setspecific()`.

The custom synchronization primitives use Pthreads mutex and condition variables. A custom `block` primitive uses `pthread_cond_wait()` to block the current thread and a custom `unblock` primitive uses `pthread_cond_signal()` to unblock threads. To support blocking and unblocking, the TCB for the **pt** target includes a mutex variable and a condition variable.

Pthreads has its own scheduler, so the client-specified scheduling code is simply ignored. Therefore, a thread system based on the **pt** target might not work as expected. For example, a client may specify application sensitive scheduling routines, e.g., LIFO queues instead of FIFO queues. Applications that expect such scheduling schemes may not work correctly when built with the **pt** target (because these schemes will be ignored).

The **pt** target is intended to allow Mezcla-based thread systems to be built on platforms that are not supported by the other target types described previously. The **pt** is especially useful for platforms that do not allow user programs to create kernel-level threads. For example, the OSF1/Alpha target is not supported by `libmp`, and consequently, `psmp`. The only way to build a Mezcla thread system on the OSF1/Alpha platform that exploits multiple processors is to use the **pt** target because Pthreads on OSF1/Alpha supports multiprocessors.

8.4 Thread Generation

The thread generation tool in the Mezcla prototype is a Perl program called `tgen` that creates specialized thread systems from a client description file and a target template. The `tgen` program is invoked as follows:

```
tgen -t <target_template> <client_description>
```

The `target_template` is one of the template file names listed in Table 8.3; target template files have a `.tpl` extension. The `client_description` is the name of a valid client description file.

The `tgen` program generates two files in the current directory (as described in Section 8.2.1) whose names are determined by the client specification file. For example, if the client specification includes `set name foothreads`, then `tgen` generates the files `foothreads.h` and `foothreads.c`.

The `tgen` program is comprised of three phases: specification parsing, active template generation, and template execution. These three phases are discussed below. (See Appendix A for a complete listing of the `tgen` source code.)

8.4.1 Phase I: Description Parsing

The first `tgen` phase parses client descriptions. A fairly simple parser based on Perl regular expressions transforms a valid client description file into an abstract syntax tree (AST). The AST is implemented as a Perl object, where each node is a Perl associative array. The AST module provides functions for adding and visiting nodes.

8.4.2 Phase II: Template Generation

The second `tgen` phase generates templates. In this phase, `tgen` transforms a target template file into an executable Perl program. The generated program is given a `.pl` extension; e.g., if `tgen` is processing `mtf_ps.tpl`, it will generate `mtf_ps.tpl.pl` as an intermediate file. The template generation phase is independent of the first phase as it depends only on the selected target template file.

Before translating the template file, `tgen` sends some initialization code to the output file. Similarly, `tgen` appends some support code to the output file once all the template lines have be translated.

`tgen` processes the selected template file so that each line is translated into Perl code. It translates lines that represent C code into `output` functions. Consider, for example, the following template fragment from the **psmp** target:

```
#include<stdio.h>
#include<stdlib.h>
#include"${name}.h"
```

It is translated to:

```
output("#include<stdio.h>\n");
output("#include<stdlib.h>\n");
output("#include\"${name}.h\"\n");
```

The `output()` function simply emits its string parameter to the current output file. Notice that this scheme accommodates Perl variable substitutions in the output strings.

The Perl annotations (listed in Section 8.3.1) in the template are expanded to larger Perl code fragments. Again, consider a template fragment from the **psmp** target:

```
@visit tcb_container
typedef struct $name {
        @perl for (@$list) { output("\t$_\n"); }
} $name;
@vend
```

It is translated to:

```
# Line 100
    if (!defined($tcb_container)) {
        print "mtf: node 'tcb_container' not defined\n";
        exit 1;
    }
    $tcb_container->visit ();
output("typedef struct $name {\n");
 for (@$list) { output("\t$_\n"); }
output("} $name;\n");
# Line 104
    mtf_ast->bye();
```

The complete source for the template parser, listed in Appendix A, shows how `tgen` translates the other template annotations.

The resulting executable template can be cached, so it does not have to be regenerated each time `tgen` is executed. It has to be regenerated only if the specific target's template has changed.

8.4.3 Phase III: Template Execution

The final `tgen` phase executes templates. In this phase, the abstract syntax tree built in Phase I is passed to the executable template created in Phase II to generate the final output source (a `.c` file and a `.h` file). Since the executable template is simple Perl code, it is executed by the Perl interpreter.

8.5 Support Libraries

The target templates presented in Section 8.2 and the resulting code generated by `tgen`, presented in Section 8.4, rely on low-level support libraries for context switching and multiprocessor operation. These libraries must be linked with the final run-time system or thread library.

8.5.1 Context Switching

The Mezcla prototype currently supports two types of context switching: direct switch and preswitch. The direct switch code comes from the original SR implementation and the preswitch code comes from QuickThreads [70].

8.5.1.1 Direct Switch

Figure 8.16 lists the interface for the direct switch library, `libds`. The interface is identical to the SR context switching code presented in Section 3.2.2.2 except for the renaming of the functions. The current implementation of the direct switch target, presented in Section 8.3.3, only uses `arg1` when initializing a context for a new TCB. Therefore, future versions of the direct switch library should remove support for the three additional arguments. This will save three words of space on the stack and approximately three word moves during TCB initialization. The `context` address, which serves as a handle to a thread context, points to the base of the stack region, i.e., the bottom of the stack. The `libds` library stores the stack pointer in a fixed offset from the base. For a description of how to use `ds_build_context()` and `ds_chg_context()`, see the presentation of the corresponding functions, `sr_build_context()` and `sr_chg_context()`, given in Section 3.2.2.2.

```
void ds_build_context(void (*entry)(), void *context, int size,
                  long arg1, long arg2, long arg3, long arg4)

void ds_chg_context(void *newctx, void *oldctx)

void ds_stk_underflow(void);
void ds_stk_overflow(void);
void ds_stk_corrupted(void);
```

Figure 8.16: Direct Switch Interface

The remaining three functions must be supplied by the code that uses `libds`, as `libds` uses these entry points as callbacks in the presence of certain error conditions. The `ds_stk_underflow()` callback is invoked when the `entry` function (also known as the start function) returns. The ds_stk_overflow() callback is invoked when ds_chg_context() detects stack overflow and the ds_stk_corrupted() callback is invoked when ds_chg_context() detects that the thread context is corrupted (if it detects that part of the context has been overwritten). Corruption is detected by placing a magic number in the thread context area.

The libds library currently supports the following processor architectures: Intel x86 (386 and higher), MIPS, Sparc, and Alpha AXP.

8.5.1.2 Preswitch

The preswitch library, libps, is essentially identical to the original QuickThreads library [70], except that minor changes have been made to the build environment to integrate QuickThreads with the Mezcla source code. The basic idea behind the preswitch approach is described in Section 5.3.2; see [70] for a complete description of QuickThreads.

```
typedef void *(qt_userf_t)(void *arg);
typedef void *(qt_prefix_t)(void *u1, void *u2, qt_userf_t *func);
typedef void *(qt_helper_t)(qt_t *oldsp, void *u1, void *u2);

void *QT_SP (void *aligned_stack, int stacksize);

void *QT_ARGS (void *sp, void *arg1, void *arg2, qt_userf_t *func,
               void (*prefix)(void *, void *, qt_userf_t *));

void  QT_BLOCK ((qt_helper_t *) helper, void *u1, void *u2,
                void *newsp);
```

Figure 8.17: Preswitch Interface

Figure 8.17 lists the QuickThreads functions used by the preswitch targets described in Section 8.3.3. The QT_SP() function is used to determine the address of the top of the stack given a pointer, aligned_stack, and the stack size, stacksize. The aligned_stack parameter points to the *beginning* of a memory region to be used as a stack; it should be aligned to meet the requirements of the target architecture. The QT_STKALIGN macro is provided by QuickThreads to determine the architecture-specific alignment.

The QT_ARGS() function is used to initialize a thread context. It is similar in functionality to ds_build_context(). The first parameter is a pointer to the top of a stack, which is usually determined by using `QT_ARGS()`. The next two parameters, `arg1` and `arg2`, are used to pass user-defined values to a *prefix* function. The fourth parameter is a pointer to a start function. The fifth parameter is a pointer to a prefix function, which is executed once to initialize the thread. It is expected that the prefix function will call the start function, `func`, with the argument, `arg1`. The use of `arg2` is user-defined, but the preswitch target types given in Section 8.3.3 use it to pass the pointer of the TCB associated with the new thread.

The `QT_BLOCK()` function is used to context switch from one thread to another. Unlike `ds_chg_context()`, `QT_BLOCK()` first switches to a *helper function* before resuming execution of the new thread. The first parameter passed to `QT_BLOCK()` is a pointer to a helper function. The next two parameters, `u1` and `u2`, are user defined values passed to the helper function. The fourth parameter, `newsp`, is the address of the stack pointer for the new thread. Unlike the direct switch interface, the handle to a thread is its stack pointer. When `QT_BLOCK()` is called, it switches to the stack of the thread pointed to by `newsp` and then executes `helper`. The *helper* function is used to execute code for the old thread on the new thread's stack; it is commonly used to save the old thread's stack pointer, `oldsp`, into a known location, such as the thread's TCB, and to either put the thread's TCB onto a queue or to clean up the old TCB if the old thread is exiting. Once the `helper` function returns, the new thread is allowed to execute.

The `libps` library currently supports the same processor architectures as `libds`: Intel x86 (386 and higher), MIPS, Sparc, and Alpha AXP.

8.5.2 Multiprocessors

To easily support a large number of multiprocessor targets, we developed a simple, operating system independent, multiprocessor interface. This interface provides basic multiprocessor functionality, including initialization, virtual processor creation, exiting, locking, and memory allocation. The resulting library is called `libmp` and it must be linked with thread systems based on the multiprocessor target.

Figure 8.18 lists the multiprocessor interface functions. The `mtf_mp_init()` function must be called before any other `libmp` function is called. It is recommended that this function be called very early in the initialization of the thread system, since many thread system functions indirectly rely on proper initialization of the multiprocessor code. The

```
/* initialization and system functions */

typedef void (*mtf_mp_vp_manager_t)(int);

int mtf_mp_init(void);
int mtf_mp_start(mtf_mp_vp_manager_t vp_manager, int count);
void mtf_mp_exit(int status);

/* processor identification */
int MTF_MP_GET_VPID(void);

/* multiprocessor locks */

int mtf_mp_lock_init(mtf_lock lock);
int mtf_mp_lock_free(mtf_lock lock);
int mtf_mp_lock_acquire(mtf_lock lock);
int mtf_mp_lock_release(mtf_lock lock);

/* malloc and free */

void *mtf_mp_malloc(size_t size);
void mtf_mp_free(void *ptr);
```

Figure 8.18: Multiprocessor Interface

psmp multiprocessor target, described in Section 8.3.3, calls `mtf_mp_init()` at the beginning of `mtf_init()`. The `mtf_mp_exit()` function is used to cleanly exit multiprocessor execution. It is called by `mtf_exit()` in **psmp**.

The `mtf_mp_start()` function is used to start new *virtual processors*. The term *virtual processor* is used because `mtf_mp_start()` usually creates a new kernel-level thread that is scheduled by the operating system and kernel-level threads may execute on different physical processors. The first parameter to `mtf_mp_start()` is a pointer to a manager function, `vp_manager`. The second parameter specifies how many virtual processes should be created. The `vp_manager` function is responsible for executing threads on the virtual processor and for executing an idle thread in the event that there are fewer threads than virtual processors. In **psmp**, a new ECB (execution control block) is associated with each virtual processor. Figure 8.19 shows the **psmp** code for `mtf_start()`, which begins multi-threaded execution.

```
int mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func,
              void *main_arg)
{
        /* initialize main tcb */
        main_tcb->next = NULL;
        main_tcb->stack = NULL;
        main_tcb->sp = NULL;

        mtf_tcb_init(main_tcb, main_func , main_arg);

        mtf_tcb_start(main_tcb);

        /* multiprocessor start */

        mtf_mp_start(mtf_mp_idle_start, MTF_MPCOUNT);

        /* we should not reach this code */
        assert(0);

        return 1;
}
```

Figure 8.19: Starting Virtual Processors

The function `MTF_MP_GET_VPID()`, which is usually implemented as a macro, returns the integer identification number of the current virtual processor. This is useful to identify the current ECB and TCB.

The `libmp` library also provides an interface to multiprocessor-safe locks. The functions support initialization and deallocation with `mtf_mp_lock_init()` and `mtf_mp_lock_free()`, respectively. It is important to free multiprocessor locks when they are no longer needed, because some implementations of `libmp` use operating system locks that consume kernel and user memory. Locks are acquired and released with `mtf_mp_lock_acquire()` and `mtf_mp_lock_release()`. In **psmp**, the `mtf_lock_*` functions are directly mapped to the equivalent `mtf_mp_lock_*` functions.

Finally, every `libmp` implementation provides interfaces for multiprocessor-safe versions of `malloc()` and `free()`. As mentioned in Section 8.1, these functions are not multiprocessor-safe on all platforms. For platforms where `malloc()` and `free()` are already multiprocessor-safe, `mtf_malloc()` and `mtf_free()` are simply macros that map to the corresponding C library functions. Otherwise, an implementation of `libmp` must explicitly protect calls to `malloc()` and `free()` with multiprocessor locks.

The framework prototype currently supports three multiprocessor platforms with `libmp`: Linux/x86, IRIX/MIPS, and Solaris/Sparc. The following sections briefly describe implementation-specific details for each of these platforms.

8.5.2.1 Linux/x86

Linux provides a system call, `__clone()`, that allows a process to create a kernel-level thread. The new kernel-level thread is called the *child*. The `__clone()` system call requires a programmer to specify a start function, a stack pointer, creation flags, and a single `void *` argument to be passed to the start function. The flags parameter specifies certain attributes that will determine how the `__clone()` system call will execute. Among other things, the flags specify what should be shared between the parent and the child (the address space, file system information, file descriptors, signal handlers, etc.). The flags also determine if the child will have the same PID as the parent or a unique PID.

Virtual processor identification could be be achieved with the `get_pid()` system call. However, system calls are somewhat expensive to be invoke on every thread operation. Therefore, `libmp` for Linux uses an x86 specific register, the *task segment descriptor*, for determining the virtual process identifier.

The Linux kernel does not have direct support for user-level, multiprocessor locks. As such, `libmp` uses x86-specific spinlocks implemented with the `xchgl` instruction. A test-and-test-and-set scheme is used.

Calling the `_exit()` system call will not kill all kernel threads associated with a process. To avoid leaving unused, but executing, kernel threads, the `mtf_mp_exit()` function for Linux must explicit kill all of the kernel threads (virtual processors) associated with the exiting kernel thread.

Finally, `malloc()` and `free()` are not multiprocessor-safe in Linux, so an explicit spinlock is used to provide mutual exclusion for memory allocation and deallocation.

8.5.2.2 IRIX/MIPS

IRIX provides direct support for kernel threads, processor identification, and multiprocessor-safe locks. Kernel threads are created with the `sproc()` system call, which requires a pointer to a start function, a flags value, and an optional `void *` argument that is to be passed to the start function. The flags value is similar in purpose to the `__clone()` flags parameter for Linux. A function, `get_tid()`, returns an integer identifier of the currently executing kernel thread. This function can be used directly for virtual processor identification.

Multiprocessor-safe locking is provided via an operating system supported spinlock interface. Unlike the spinlocks used for the Linux implementation of `libmp`, the IRIX locks allow a kernel thread to block if it spins too long. The Linux locks will spin until the lock is acquired.

The `malloc()` and `free()` functions are multiprocessor-safe in IRIX.

8.5.2.3 Solaris/Sparc

Similar to IRIX, Solaris provides direct support for kernel-level threads through the Solaris threads interface. The Solaris threads interface is similar to the Pthreads interface, except Solaris threads allow a programmer to explicitly create kernel-level threads. The `thr_create()` function is used to create a new virtual processor. Its parameters are similar to those for `__clone()` and `sproc()`.

Virtual process identification is achieved with the `thr_self()` function and locking is achieved with the `thr_mutex_*` functions. The `thr_mutex_*` functions work just like the `pthread_mutex_*` functions. The `malloc()` and `free()` functions are multiprocessor-safe in Solaris.

Solaris also offers a direct LWP (light-weight process) interface, which is similar to IRIX's support for kernel-level threads. A future version of `libmp` for Solaris will be implemented directly in terms of the LWP interface.

8.6 Summary

This chapter presented a prototype implementation of the Mezcla thread framework, which captures most of the design requirements presented in Chapter 6. Specifically, the implementation provides a client description format, a target description format in terms of target templates, and a tool called `tgen`. The `tgen` tool, implemented in Perl, parses client specifications and then blends the client-supplied properties with a particular target specification to produce a specialized thread system. The implementation is used for all of the experimental systems presented in Chapter 9.

Chapter 9

Experimental Results

This chapter describes several experiments designed to evaluate the performance and usability of the Mezcla prototype described in Chapter 8. It presents experimental results from three different thread systems built using Mezcla: a simple semaphore-based thread library called SemThreads, a thread system for the SR run-time system, and a thread system for the Kaffe Java virtual machine [116]. While the SemThreads and SR implementations are relatively complete, the Kaffe implementation is preliminary and can execute only very simple Java programs but still demonstrates potential performance gains. For each experimental system, we present an overview of the implementation and performance results for microbenchmarks and medium-sized applications.

We built and executed our benchmark programs on four target platforms: Alpha-OSF1, MIPS-IRIX, PII-Linux, and Sparc-Solaris. Table 9.1 lists the specifications for each of these targets.

Platform	Architecture	Clock Speed	Processors	OS	Compiler
Alpha-OSF1	Alpha 21164A	400MHz	2	OSF1 V4.0	GCC 2.8.1
MIPS-IRIX	MIPS R10000	180MHz	2	IRIX 6.5	GCC 2.8.1
PII-Linux	Pentium II	450MHz	2	Linux 2.2.3	GCC 2.7.2.3
Sparc-Solaris	UltraSparc II	360MHz	2	SunOS 5.7	GCC 2.8.1

Table 9.1: Target Platform Specifications

For each platform, we built a Mezcla thread system for each supported target type presented in Section 8.3.3 and listed again in Table 9.2. The Mezcla target **psmp** is not supported on Alpha-OSF1 because OSF1 does not provide a direct system call for creating kernel-level threads. For the **psmp** target type and some of the benchmarks, we ran each benchmark twice: once with a thread system using only one processor (**psmp1**) and once with a thread system using two processors (**psmp2**). The purpose of **psmp1** is to show the additional cost of locking and multiprocessor support over the uniprocessor targets (**ds**, **ps**, and **pspr**). In general, the Pthreads target type, **pt**, uses as many processors as available (two for all of our targets).

Target	Description
ds	Direct Switch
ps	Preswitch
pspr	Preswitch Preemption
psmp1	Preswitch Multiprocessor (1 processor)
psmp2	Preswitch Multiprocessor (2 processors)
pt	Pthreads

Table 9.2: Target Types

Each microbenchmark described in this chapter measures the cost of an operation by executing the operation n times in succession. To compute the cost of a single operation, we divide the total benchmark time by n. However, this measurement includes the overhead introduced by the loop that drives the benchmark. For each Mezcla-based system, we provide measurements of the loop overhead. We ran each benchmark several times on lightly loaded systems. The variances from run to run were very small. In the following sections, when we refer to the "results of a benchmark" we mean the execution time of running the benchmark. The former phrase is used for brevity.

Most of the results presented in the following sections are given in microseconds unless otherwise specified. In addition, the symbol **NA** (not available) is used in the result tables to denote that the value for an experiment was not obtained. Such experiments are not represented in the graphs. The reasons for **NA** results are explained on a per benchmark basis. However, as noted above, the **psmp** target is not supported on the Alpha-OSF1 platform. So, **NA** is recorded in the result cells of the benchmark tables for **psmp** and Alpha-OSF1 combinations. Also, due to time constraints, we were unable to track down the causes for some of the failed benchmarks. Again, such failures are identified on a case by case basis. The overall number of failed experiments is extremely small compared to the number of successful experiments, so they do not diminish the results or the implications of the results presented in this chapter.

9.1 SemThreads: A Simple Thread System

To facilitate the development of the Mezcla prototype and demonstrate how to use Mezcla for library-based thread systems, we developed a simple thread system called *SemThreads*. SemThreads is similar to MutexThreads from Chapter 7, except that SemThreads has more sophisticated forms of synchronization.

The SemThreads library supports a small set of operations for thread creation and synchronization. The primary method of thread synchronization is accomplished with semaphore variables (see Section 2.1.2). Threads may also synchronize using a "join" operation. The rest of this section presents the SemThreads Interface, the Mezcla implementation of SemThreads, and some experimental results.

9.1.1 Interface

The SemThreads interface supports two data types and a limited set of thread and semaphore operations: create, join, exit, P, and V. The data types are `sem_tcb` and `sem`. The `sem_tcb` data type is used to identify different threads; it is produced by the Mezcla thread generation tool (see Section 9.1.2). The `sem` data type represents semaphores, which are used as input parameters to the synchronization operations. Figure 9.1 lists the data types and functions supported by the SemThreads interface.

The `sem_start()` function is used to initiate threaded execution. It is passed a function pointer, `func`; a single `void *` argument, `arg`; and an integer identifier, `id`, to be associated with the start thread. The `sem_start()` function initializes the thread system and begins execution with `func`, which is passed `arg`. `sem_start()` does not return to the caller. An alternative version of `sem_start()` is `sem_istart()`, which performs initialization and does return to the caller. It too is passed an identifier argument, `id`, which is associated with the current thread. The `sem_istart()` function is used to support implicit initialization, whereby the main thread can be used as a normal thread. (See Section 6.4.2.1 for a discussion of explicit versus implicit initialization.)

Threads are created with `sem_create()`[1]. This function is passed a pointer to a

[1]In retrospect, using `sem` as the prefix was a poor choice. For example, someone not familiar with the

```
#include "semgen.h"

/* data types */

typedef struct sem {
        int value;
        int blocked;
        mtf_lock lock;
        sem_container container;
} sem;


/* functions */

void sem_start(mtf_tcb_func *func, void *arg, int id);
void sem_istart(int id);

void sem_create(sem_tcb **tcb, mtf_tcb_func *func, void *arg, int id);
void sem_join(sem_tcb *tcb, void **join_value);
void sem_exit(void *value);
int  sem_getid(void);

void sem_init(sem *s, int value);
void sem_destroy(sem *s);
void sem_P(sem *s);
void sem_V(sem *s);
```

Figure 9.1: The SemThreads Interface

`sem_tcb` pointer; a function pointer, `func`; a `void *` argument, `arg`; and an integer identifier, `id`. The `sem_create()` function creates a new thread and returns a pointer to the new thread's TCB in the `tcb` pointer, which is passed by reference. The new function is called with argument `arg`. The `id` argument is a user-defined identifier that will be associated with the new thread (it can be accessed subsequently with `sem_getid()`). For a non-preemptive uniprocessor target, the new thread will not begin execution until the creating thread blocks or exits. For all other targets (preemptive uniprocessor and multiprocessor targets), the new thread can begin execution anytime after it has been created.

A thread must exit using `sem_exit()`, which takes a `void *` return value as a parameter. The `sem_exit()` function terminates the calling thread and releases any memory associated with the thread (e.g., the stack and TCB). The `sem_exit()` function also works in concert with `sem_join()` to synchronize threads: a thread calls `sem_join()` to wait for the specified thread to exit. Consequently, a call to `sem_exit()` may wake up another thread waiting to join with the exiting thread. A call to `sem_join()` on a thread that has exited does not block. The `sem_join()` function takes two arguments: a pointer to a `sem_tcb` and a pointer to a `void *` return value. The exiting thread's return value is passed to the joining thread via the `join_value` argument.

The `sem_getid()` is used to access the user-defined identifier associated with the current thread.

In addition to `sem_exit()` and `sem_join()`, threads can also synchronize with semaphores. To initialize a semaphore, a programmer provides `sem_init()` with a pointer to a `sem` variable (which must be preallocated) and an initial value. Semaphores are destroyed with `sem_destroy()`. The two main operations on semaphore variables are `sem_P()` and `sem_V()`, which correspond to standard **P** and **V** semaphore operations.

9.1.1.1 Example Program: Dining Philosophers

To illustrate how to use the SemThreads interface, Figures 9.2 and 9.3 list the source code for a SemThreads version of the dining philosophers problem.

Figure 9.2 contains the startup code. Threaded execution begins in `main()`. The call to `sem_start()` begins `philosopher_main()`. In turn, `philosopher_main()` initializes the `forks` array of semaphores. It then creates `N` philosophers. Finally,

interface might assume that `sem_create()` creates a semaphore, not a thread.

```
#include <stdio.h>
#include "semthreads.h"

#define N 5
sem forks[N];
int iters;

void philosopher_main(void *arg)
{
        int i;
        sem_tcb *t[N];

        for (i = 0; i < N; i++) {
                sem_init(&forks[i], 1);
        }

        for (i = 0; i < N; i++) {
                sem_create(&t[i], (mtf_tcb_func *) philosopher,
                           (void *) i, 0);
        }

        for (i = 0; i < N; i++) {
                sem_join(t[i], NULL);
        }

        mtf_exit(0);
}

int main(int argc, char **argv)
{
        if (argc != 2) {
                printf("usage: %s <iters>\n", argv[0]);
                exit(1);
        }
        iters = atoi(argv[1]);

        sem_start((mtf_tcb_func *) philosopher_main, (void *) 0, 0);
        return 0;
}
```

Figure 9.2: Dining Philosophers in SemThreads, Part I

`philosopher_main()` waits for each philosopher to finish executing by attempting to join with each philosopher thread.

```
void philosopher(void *arg)
{
        int id = (int) arg;
        int i, left, right;

        if (id == 0) {
                right = id;
                left  = (id + 1) % N;
        } else {
                left = id;
                right = (id + 1) % N;
        }

        for (i = 0; i < iters; i++) {
                sem_P(&forks[left]); sem_P(&forks[right]);
                printf("Philosopher %d eating\n", id);

                sem_V(&forks[left]); sem_V(&forks[right]);
                printf("Philosopher %d thinking\n", id);
        }

        sem_exit(NULL);
}
```

Figure 9.3: Dining Philosophers in SemThreads, Part II

Figure 9.3 contains the code for the `philosopher()` function. The standard precaution of switching the fork acquisition order for one of the philosophers is used to prevent deadlock. The main loop of the `philosopher()` function simply acquires two forks to eat and then releases the forks to begin thinking. This loop is repeated `iters` times.

9.1.2 Mezcla Implementation

The SemThreads interface is implemented with the Mezcla thread framework prototype presented in Chapter 8. Using the client specification format, it specifies a custom TCB, a custom container, and two sets of custom blocking primitives. Figures 9.4 and 9.5 list the Mezcla client description file for SemThreads (`semgen.mtf`).

```
# semgen.mtf
#
# Mezcla client description file for SemThreads

# Preamble

set name semgen
set client_inc semclient.h

# Data

data tcb sem_tcb {
        int id;
        int exited;
        struct sem_tcb *join_tcb;
        void *join_value;
        struct sem_tcb *next;
        mtf_lock lock;
}

data tcb_container sem_container {
        struct sem_tcb *head;
        struct sem_tcb *tail;
}
```

Figure 9.4: The Mezcla Client Description for SemThreads, Part I

The SemThreads TCB contains client specific data for custom scheduling and synchronization. The `id` variable is used to associate a user-defined integer identifier with a thread; it is returned by `sem_getid()`. The `exited` variable is used by `sem_join()` to determine if a thread has exited. Its initial value is 0 and it is set to 1 by `sem_exit()`. Both `join_tcb` and `join_value` are used for joining threads. A thread's `join_tcb` field is used to point to the TCB of the thread blocked waiting for the thread to exit. The `join_value` variable is used to pass a thread's exit value, as specified with `sem_exit()`, to a joining thread.

The `next` pointer is used to link TCBs inside TCB containers. SemThreads uses a simple, single linked list data structure for the TCB containers. Therefore, only a `next` pointer is needed. More complicated container types may require more fields.

An `mtf_lock` variable, `lock`, is specified to adhere to the Mezcla locking conventions. As described in Chapter 8, such a lock is necessary to ensure proper synchronization on preemptive, multiprocessor, and Pthreads targets.

```
# Code

code sched sem_container {
        set init sem_container_init
        set put  sem_tcb_put
        set get  sem_tcb_get
        set head sem_tcb_head
}

codel sync sem_container {
        block sem_block {
                sem_tcb_put(b_tcb, b_container);
        }
        unblock sem_unblock {
                b_tcb = sem_tcb_get(b_container);
        }
}

codel sync sem_tcb {
        block join_block {
                b_container->join_tcb = b_tcb;
        }
        unblock join_unblock {
                b_tcb = b_container->join_tcb;
        }
}
```

Figure 9.5: The Mezcla Client Specification for SemThreads, Part II

The client specification file from Figures 9.4 and 9.5 is given as input to the Mezcla thread generation tool, `tgen`, to generate specialized primitives for SemThreads. Recall from Section 8.4 that `tgen` allows a programmer to specify a client description file and a desired target type. The specialized primitives are used by the SemThreads library to implement the SemThreads interface listed in Figure 9.1. See Appendix C for listings of the client supplied portion of the SemThreads implementation (`semgen.mtf`, `semclient.h`, `semqueues.h`, `semthreads.h`, and `semthreads.c`).

9.1.3 Experiments

Now we look at a series of benchmarks used to measure the absolute performance of several SemThreads operations. These benchmarks give insight into the performance of thread systems built using Mezcla and also the relative performance of each Mezcla target. Table 9.3 lists the benchmarks we use in the following sections.

Benchmark	Description
loop	Loop overhead (included in all benchmarks below)
function	Function call overhead
intadd	Integer addition
floatadd	Floating point addition
syscall	System call overhead
threadid	Thread identification
forkjoin	Thread creation and termination
sync1	Semaphore synchronization without context switching
sync2	Semaphore synchronization with 2 semaphores and context switching
sync3	Semaphore synchronization with 1 semaphore and context switching
pvsieve	Parallel segmented sieve for finding prime numbers
barnes	Barnes-Hut N-body Simulation (from Splash-2 benchmarks)

Table 9.3: SemThreads Benchmarks

9.1.3.1 Platform Benchmarks

The first five benchmarks from Table 9.3 measure the cost of several simple operations to help give a picture of basic platform performance. These benchmarks are by no means complete; they are intended to help indicate relative system performance for our platform targets. Table 9.4 and Figure 9.6 present the results of the platform benchmarks.

Benchmark	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
loop	0.0572	0.0447	0.0124	0.0389
function	0.0915	0.1395	0.0270	0.0528
intadd	0.0932	0.0503	0.0175	0.0501
floatadd	0.1152	0.0727	0.0623	0.0557
syscall	0.7008	2.8267	0.6780	1.3227

Table 9.4: Platform Performance (in microseconds)

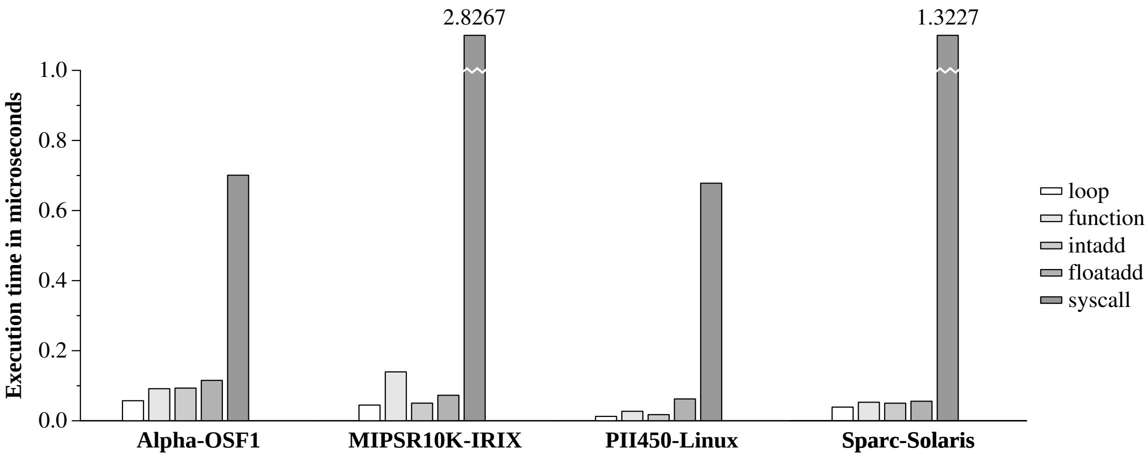

Figure 9.6: Platform Performance

The `loop` benchmark measures the loop overhead cost per iteration that is included in all of the SemThreads microbenchmarks. The next three platform benchmarks, `function`, `intadd`, and `floatadd`, measure the costs of function call, integer addition, and floating point additions, respectively. The final benchmark, `syscall`, measures the cost of the `getpid()` system call, which indicates the cost of entering into and exiting from the kernel.

The only interesting result from these benchmarks is that the function call result for MIPS-IRIX is relatively high compared to the other platforms. This could affect the relative performance of thread operations that require one or more function calls.

9.1.3.2 Thread Identification

The thread identification benchmark measures the cost of determining the currently running thread. In SemThreads, the `sem_getid()` function is provided to identify the current thread; it is implemented using the Mezcla primitive for thread identification. The cost of thread identification for uniprocessor target types is low because it simply dereferences a global pointer to the current thread. However, for **psmp**, a more sophisticated implementation is needed because a target may have more that one processor executing threads simultaneously. In this case, a simple global pointer will not work. Multiprocessor thread identification is implemented using platform-specific processor identification mechanisms. (See Section 8.5.2 for a description of how multiprocessor thread identification is implemented in `libmp` for each test platform.) For the Pthreads target type, **pt**, the thread identification benchmark measures the cost of the `pthread_getspecific()` function.

Thread identification is used pervasively in the generated thread system to determine the currently running thread so that a blocking or yielding thread can be put into the appropriate container. Thus, it is used by almost every SemThreads function and most of the Mezcla generated primitives. In addition, correct processor identification is needed to update any state related to the ECB (execution control block) for a virtual processor. For example, **psmp** associates an idle thread with each ECB. When there are no more threads to run, the virtual processor must run a specific idle thread pointed to by the ECB. The correct ECB is located using processor identification. Therefore, the expense of thread identification will have a large impact on the performance of many Mezcla primitives.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
ds	0.0748	0.0559	0.0133	0.0556
ps	0.0923	0.0523	0.0250	0.0556
pspr	0.0606	0.0515	0.0159	0.0559
psmp1	NA	0.0670	0.0244	0.1139
psmp2	NA	0.0670	0.0244	0.1141
pt	0.1615	0.1455	0.0984	0.2158

Table 9.5: Thread Identification (in microseconds)

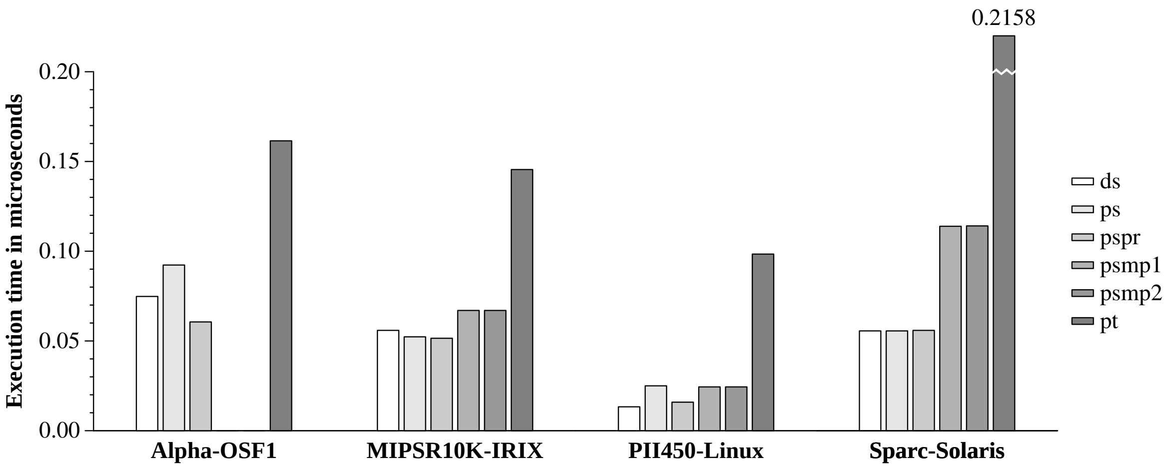

Figure 9.7: Thread Identification

Table 9.5 and Figure 9.7 present the results of the thread identification benchmark. Notice that the execution times of the **psmp** target types are proportionally more expensive than the uniprocessor targets on the same platform. Also, the thread identification benchmark shows that all platform-specific Pthreads implementations of `pthread_getspecific()` are rather expensive. Just as thread identification affects the performance of all the Mezcla primitives, it is likely that thread identification also affects the performance of many Pthreads functions. This may partially explain (as will be shown in the following sections) why basic thread operations in Pthreads are comparatively expensive.

9.1.3.3 Create/Join

The create/join benchmark determines the cost of thread creation, exiting, and joining (synchronization). This benchmark creates a thread using `sem_create()` and waits for the thread to exit using `sem_join()`. The created thread simply exits. Table 9.6 and Figure 9.8 present the results for the create/join benchmark. The cost of create/join for the Pthreads target type is significantly higher than for the other target types; therefore, the bars for the Pthreads targets are broken in Figure 9.8.

In general, the cost of create/join increases with the relative functionality of the target types. For example, **pspr** is more expensive than **ps** because of the cost of preemption, which occurs many times during the execution of this benchmark. The absolute cost of a single create/join operation in **pspr** is not much higher than create/join for **ds** and

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
ds	13.4380	5.0094	1.7571	7.1513
ps	10.7398	4.7958	1.7464	5.9039
pspr	17.7750	6.3579	2.0261	6.6907
psmp1	NA	10.4872	3.2582	9.2185
psmp2	NA	13.7212	3.3654	7.9565
pt	132.3288	35.1852	206.0510	52.1936

Table 9.6: Create/Join (in microseconds)

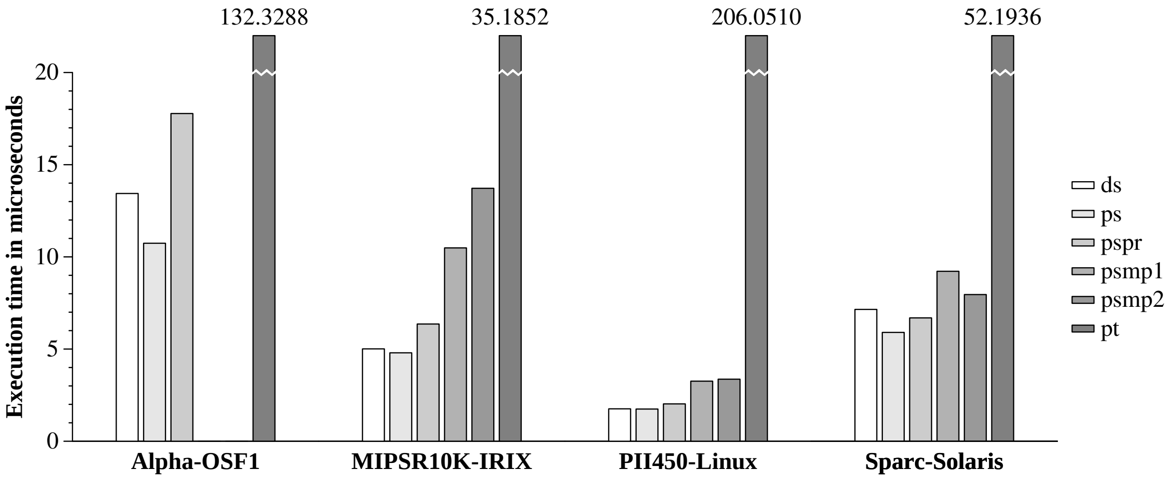

Figure 9.8: Create/Join

ps; however, create/join for **psmp** is more expensive than it is for the uniprocessor target types.

The PII-Linux time for create/join with the Pthreads target is proportionally more expensive than on the other test platforms because the Linux implementation of Pthreads is built on Linux kernel threads, whereas the other platforms have hybrid implementations of Pthreads that use both user-level and kernel-level threads.

9.1.3.4 Synchronization

The next three benchmarks, `sync1`, `sync2`, and `sync3`, measure the cost of various types of thread synchronization. Each benchmark uses the semaphore functions, `sem_P()` and `sem_V()`, for synchronization. The `sync1` benchmark (see Table 9.7 and Figure 9.9) measures the cost of executing `sem_P()` and `sem_V()` in a single thread without blocking.

The `sync2` benchmark (see Table 9.8 and Figure 9.10) uses two semaphores to force two threads to strictly alternate, thus forcing a context switch on uniprocessor target types. For **psmp** and **pt**, the two threads may be running on separate processors, so the semaphores simply regulate their execution. Finally, the `sync3` benchmark (see Table 9.9 and Figure 9.11) creates two threads that compete for a single semaphore.

Similar to the results from the create/join benchmark, the cost of thread synchronization increases with an increase in target type functionality. Also, a similar jump in execution time occurs when comparing the uniprocessor target to **psmp**. For some of the test platforms, the performance decreases slightly when using **psmp** with two processors instead of one processor. This is partially due to extra contention for the global ready container introduced by having two virtual processors.

The tables show that for sync1, the **pt** target on Sparc-Solaris performs reasonably well compared to the other targets. However, once true synchronization is required, as in `sync2` and `sync3`, the **pt** target performs quite poorly.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
ds	0.0799	0.3084	0.0534	0.1218
ps	0.0799	0.3069	0.0538	0.0780
pspr	0.1740	0.4764	0.0963	0.1589
psmp1	NA	1.2217	0.3208	0.4810
psmp2	NA	1.2191	0.3193	0.4843
pt	0.7491	1.7898	0.8405	0.6651

Table 9.7: sync1: Synchronization without Context Switching (in microseconds)

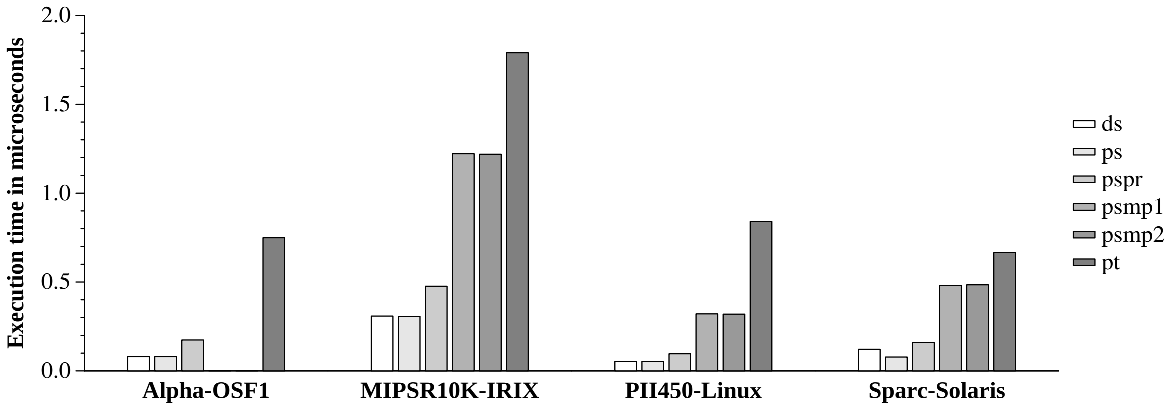

Figure 9.9: sync1: Synchronization without Context Switching

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
ds	0.8691	2.3103	0.5312	3.2860
ps	0.8062	2.3814	0.5753	2.8395
pspr	1.3348	3.4021	0.7422	3.2298
psmp1	NA	7.0080	1.8986	5.2872
psmp2	NA	7.0481	2.2851	5.1851
pt	18.5222	17.2500	44.1097	17.6236

Table 9.8: sync2: Synchronization with Context Switching (in microseconds)

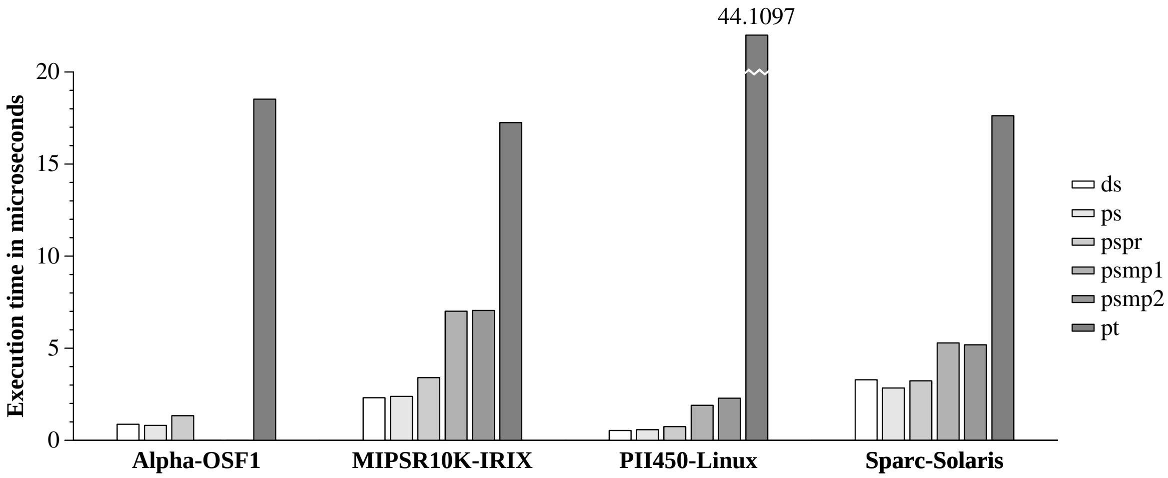

Figure 9.10: sync2: Synchronization with Context Switching

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
ds	1.0642	2.5632	0.5250	3.4163
ps	0.8791	2.6250	0.5367	3.0299
pspr	2.1326	3.6482	0.7892	3.4791
psmp1	NA	8.0966	2.1824	5.5958
psmp2	NA	8.1391	2.3356	5.5844
pt	22.9904	19.8729	95.0294	30.4863

Table 9.9: sync3: Synchronization with Context Switching (in microseconds)

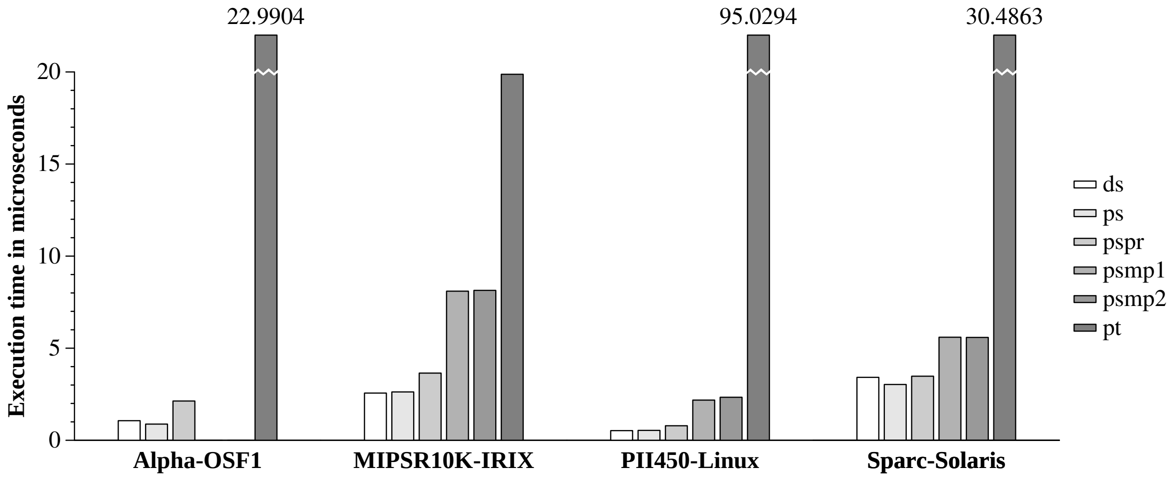

Figure 9.11: sync3: Synchronization with Context Switching

9.1.3.5 Parallel Sieve

The first application benchmark is a coarse-grain, "embarrassingly" parallel version of the Sieve of Eratosthenes for finding prime numbers. This implementation divides the range of numbers (0 to n) to be searched into segments, where the size of each segment is a multiple of $\sqrt{n}$. Initially, all of the primes in the base segment, $[0 \ldots \sqrt{n}]$, are computed. The rest of the segments can be "sieved" independently, in parallel, using the primes found in the base segment. The independent threads require no synchronization; the only synchronization that is needed is a barrier that waits for all the threads to complete. The barrier is implemented using `sem_join()`. For the uniprocessor target types, only a single thread is used.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
ds	19.8974	17.3191	9.0641	11.6797
ps	19.1708	17.4151	9.0601	14.8572
pspr	19.2378	17.2312	9.0882	11.6943
psmp2	NA	10.3000	4.5815	5.8795
pt	14.8879	10.0700	4.5820	7.3696

Table 9.10: Parallel Segmented Sieve (in seconds)

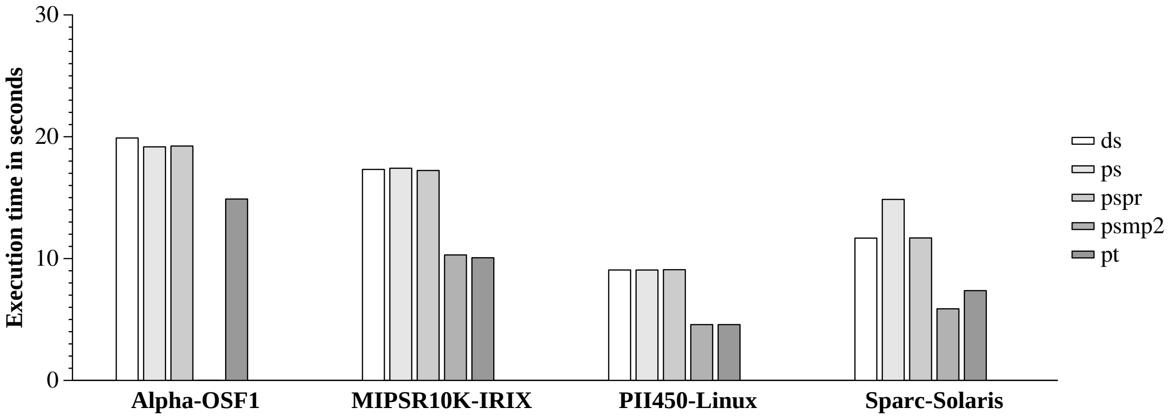

Figure 9.12: Parallel Segmented Sieve

Table 9.10 and Figure 9.12 present the results of the parallel sieve benchmark. Unlike the previous microbenchmarks, the results for **psmp** using one processor (**psmp1**) are not included since this combination simply would not be used for real applications. In general, the multiprocessor targets exhibit almost linear speed-up when two processors are used. Notice that since the running time is dominated by computation, the various target types have similar execution times. The **pt** target on Alpha-OSF1 and Sparc-Solaris does not achieve as much speed-up as does the **pt** target on MIPSR10K-IRIX and PII450-Linux. This might be the result of different scheduling policies found in the different Pthreads implementations.

This benchmark also illustrates that for coarse-grain parallel code, the individual costs of the thread operations such as `sem_create()` are not significant. For example, the **pt** target performed significantly worse on all the previous microbenchmarks, but performs comparably to the other target types for parallel sieve.

9.1.3.6 Barnes-Hut N-body Simulation

The next application benchmark is the Barnes-Hut n-body simulation from the Splash-2 benchmark suite [118]. The algorithm can use n threads (or processors) for the simulation. The simulation uses the default input parameters.

Table 9.11 and Figure 9.13 present the results of the Barnes-Hut benchmark. This benchmark shows how the cost of thread primitives can adversely affect the running time of a parallel application that requires medium to fine grain synchronization. The table and figure show that when using two processors, the results for **psmp2** on PII450-Linux exhibits slightly less than linear speed-up, while **psmp2** on MIPSR10K-IRIX exhibits super-linear speed-up. We believe this is due to better cache behavior for the **psmp2** executable.

The **pt** targets did not function properly due to incompatibilities between the Barnes-Hut code and Pthreads. Further investigation is required to understand the cause of the incompatibilities. The benchmark also failed to run with the **psmp2** target on the Sparc-Solaris target (we did not track down the cause of this problem due to time constraints).

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
ds	22.1337	18.8294	16.6446	12.3244
ps	27.6950	19.2801	17.1084	12.1803
pspr	25.2616	18.7888	15.8746	12.1800
psmp2	NA	7.3704	8.4207	NA
pt	NA	NA	NA	NA

Table 9.11: Barnes-Hut N-body Simulation (in seconds)

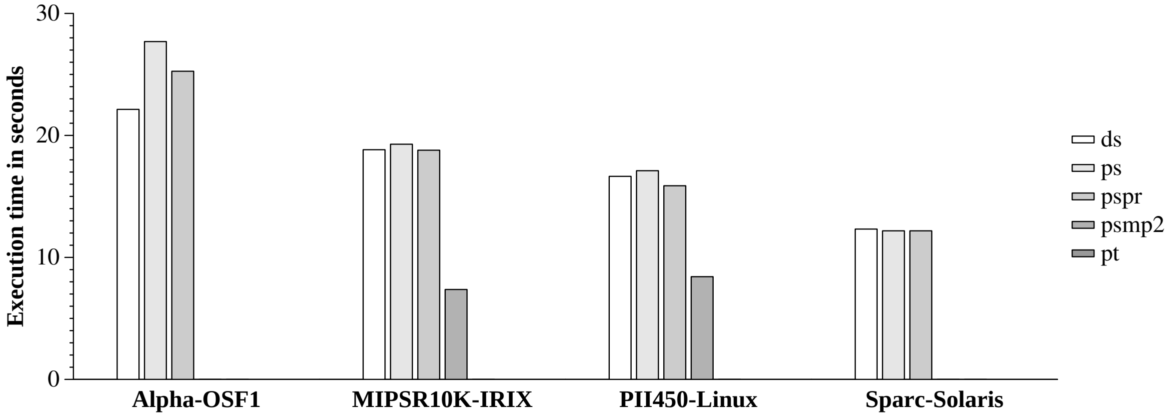

Figure 9.13: Barnes-Hut N-body Simulation

9.1.4 Discussion

The SemThreads benchmarks were designed to show the absolute cost of thread operations implemented with Mezcla generated thread primitives and to show the relative costs of the different target types. The results generally show that more complicated targets have higher costs for thread operations and that the multiprocessor targets have higher costs than the uniprocessor targets due to low-level locking and thread identification.

The absolute difference in performance between the **ds** and **ps** target types was not as large as expected, so the advantage of using **ds** over **ps** on uniprocessors is not clear. However, we did not implement a **ds**-based multiprocessor target due to the reasons described in Section 5.3.2. So, it seems that **ds** could be eliminated as a target type from the Mezcla prototype.

The **pt** target was implemented to support platforms that already have a Pthreads implementation, but do not have a Mezcla target template. The **pt** target is also useful for platforms, such as Alpha-OSF1, that do not provide kernel interfaces for implemented virtual processors (such interfaces are needed by `libmp`). However, the SemThreads benchmarks show that using Pthreads results in expensive thread operations. The high cost is partially due to the extra code generated by Mezcla to map the Mezcla primitives to Pthreads functions, but most of the cost comes from the actual Pthreads implementations themselves.

One of the most important results of the SemThreads benchmarks is that it is clear that thread system specialization based on the target type can help improve thread performance. For example, as seen in the sync2 benchmark (Table 9.8 and Figure 9.10), **pspr** is 13% to 66% slower than **ds** and **ps**, and **psmp** is 2 to 4 times slower than **ds** and **ps**. Therefore, picking the right target can greatly improve synchronization performance. A single thread system implementation, such as Pthreads, performs poorly for all target types, whereas specialized thread systems can be optimized if the target type is known.

9.2 SR

To demonstrate the utility of Mezcla for implementing concurrent programming languages, the thread system in the original SR implementation was replaced with a Mezcla-based thread system. Unlike the SemThreads experiments, using SR facilitates performance comparisons of Mezcla-based run-time systems to custom run-time systems. This section describes the Mezcla implementation and some experimental results that compare the performance of the original SR run-time system to Mezcla-based SR run-time systems.

9.2.1 Mezcla Implementation

Recall from Section 3.2 that the original version of SR supports both uniprocessors (in standard SR) and multiprocessors (in MultiSR). Both implementations use a custom user-level thread system. For the Mezcla implementation of SR, we replaced the low-level thread code and multiprocessor support in the original implementation with Mezcla primitives and a Mezcla generated TCB. See Appendix D for a listing of the Mezcla client description file for SR.

With this implementation, we have attempted to integrate Mezcla as tightly as possible into the SR run-time system. The Mezcla primitives serve as the layer of portability for threads in the SR run-time system. While this implementation reuses as much of the original run-time system as possible, a "pure" Mezcla implementation would probably have a different design. This section outlines the major modifications made to the original SR run-time system. (We omit a discussion of the "cosmetic" changes to the SR build environment.)

- **The SR TCB** The original SR implementation provides the notion of a TCB through a structure called `proc_st`. The Mezcla client TCB description leverages off this structure by including the `proc_st` fields. Appendix D lists the complete client description file (`srgen.mtf`) for the Mezcla implementation of SR. In addition, the TCB data type name is also `proc_st`. This allows the reuse of much of the original run-time system code that depends on the `proc_st` name.

  In order to reduce the overhead associated with allocation and deallocation of TCBs and thread stacks, the Mezcla implementation uses the *pool* mechanism found in the original run-time system. (The run-time system memory pool allows common structures to be recycled instead of always using the standard `malloc()` and `free()` library functions.) The Mezcla implementation uses custom allocation and deallocation call-back functions that are passed to `mtf_init()` during Mezcla initialization.

- **The SR ECB** The original MultiSR run-time system used "job servers" to support multiprocessors. Essentially, a job server is created for each processor. A job server structure, `private_st`, is used to store processor-specific state. The most important use of `private_st` is for supporting iteration counting (to prevent an SR process from dominating the CPU) and recursive locking of the global ready queue lock. The client description of the ECB includes most of the `private_st` fields. Similar to the Mezcla TCB, we specified `private_st` to be the Mezcla generated name for the ECB data type.

- **Locking and Memory Allocation** We used the MultiSR infrastructure to support Mezcla initialization and the Mezcla locking and memory allocation primitives. Specifically, the new MultiSR target, `mtf`, defines MultiSR locking and memory allocation using Mezcla primitives (see Figure 9.14).

```
#define MALLOC(a)       mtf_malloc(a)
#define UNMALLOC(a)     mtf_free(a)

#define multi_mutex_t mtf_lock

#define multi_lock(a)        mtf_lock_acquire(a)
#define multi_unlock(a)      mtf_lock_release(a)
#define multi_reset_lock(a)  {\
        mtf_lock_free(a);     \
        mtf_lock_init(a);     \
}
#define multi_alloc_lock(a)  mtf_lock_init(a)
#define multi_free_lock(a)   mtf_lock_free(a)
```

Figure 9.14: Mezcla SR Locking and Memory Allocation

- **SR Process Creation Arguments** The original SR run-time system allows an initial function for a new thread to be passed up to four arguments. However, Mezcla allows only a single argument to be passed to the initial thread function. To support four arguments, the Mezcla implementation uses the intermediate function technique

(Section 4.2.1.4); it adds four argument fields and a function pointer to the TCB. These fields are initialized by the SR run-time system during a process create operation. An intermediate function becomes the initial thread function. It is passed a single argument, which is a pointer to the new thread TCB. The intermediate function calls the "real" initial function with the arguments from the TCB (see Figure 9.15).

```
int sr_intermediate(void *arg)
{
        Proc pr = (Proc) arg;

        (pr->func)(pr->arg1, pr->arg2, pr->arg3, pr->arg4);
}
```

Figure 9.15: SemThreads TCB Container Description

- **SR Process Killing** The original SR implementation supports process killing during the destruction of resources. If the **destroy** statement is used on a resource that has running threads, those threads will be killed. Because the Mezcla prototype does not support thread killing, Mezcla SR disables process killing. This means that processes can terminate only by exiting and that processes cannot destroy resources containing active processes.

- **SR Semaphores** The original SR run-time system implements semaphores that are used both to support the language-level semaphores and for internal synchronization. We modified the semaphore implementation to use the new Mezcla custom primitives. Due to the locking hierarchy defined in the original SR run-time system, the semaphore implementation was overly conservative with respect to the global ready queue lock. Mezcla obviates the need for the ready queue lock to be visible to the run-time system (because Mezcla implements its own ready queue lock), and allows a semaphore operation to first acquire the semaphore lock itself. This change slightly improves the performance of **P** operations that do not block and **V** operations that do not unblock a waiting thread.

- **Invocation Queue Processing** The original SR run-time system make extensive use of the "decoupled" approach to blocking threads (see Section 6.4.3.1) during invocation queue processing. However, Mezcla does not currently support decoupled blocking. Therefore, we converted all of the decoupled blocking to coupled blocking.

- **SR Compiler Generated Thread Identification** The original SR compiler generated MultiSR-specific thread identification code. We modified the SR compiler to generate the Mezcla primitive for thread identification (i.e., `mtf_tcb_current()`).

- **I/O Blocking** Because Mezcla does not support blocking I/O and threads, the modified SR run-time system does not support blocking I/O. The consequence is that SR applications that make a blocking I/O call will block all other SR processes until the I/O request is serviced. For the **psmp** target, a blocking process will only block one CPU. For the **pt** targets, a blocking process will not block other processes because all of the Pthreads implementations support blocking I/O. (This is another benefit to using the **pt** target.)

- **Deadlock Detection** We also removed deadlock detection support from the original SR run-time system. The consequence is that SR programs that rely on deadlock detection for termination may not terminate. (Adding deadlock detection to the Mezcla implementation of SR could be accomplished using the technique described in [94]. It was not implemented due to time constraints.)

The functionality not supported in the Mezcla implementation of SR (i.e., process killing, I/O blocking, and deadlock detection) is not used by our benchmark programs. In addition, the unsupported functionality is not on the critical path of the standard SR synchronization mechanisms. Therefore, the Mezcla implementation does not have a performance benefit over the original SR implementation due to unsupported features.

9.2.2 Experiments

This section presents a series of benchmarks used to measure performance of the original SR run-time systems compared to the performance of Mezcla-based SR run-time systems. The test platforms are the same as those used for SemThreads (See Table 9.1 in Section 9.1.3). In addition, we have built a Mezcla-based SR run-time system for each Mezcla target type listed in Table 9.2. The test programs consist of a set of microbenchmarks used to evaluate SR's concurrency mechanisms [13] (the same set of benchmarks were used in Chapter 4 for evaluating early experimental run-time systems). We also measure the performance of parallel matrix multiply.

In the following tables, **std** represents the standard, uniprocessor version of SR and **multi** represents MultiSR. All the versions of SR were compiled with optimizations turned on and debugging support removed. The experiments using the **psmp** target are configured to use two processors (similar to the **psmp2** target from the SemThreads microbenchmarks from Section 9.1.3). MultiSR and **psmp** are not supported on the Alpha-OSF1 platform due to the lack of direct kernel-thread support in OSF1. In addition, we experienced problems running some of the benchmarks on the MIPSR10K-IRIX platform. Further investigation is needed to uncover the cause of these problems.

9.2.2.1 Loop Overhead

This benchmark measures the loop overhead that occurs in all of the SR benchmarks; it is similar to the loop overhead benchmark used in the SemThreads experiments (see Section 9.1.3.1). Table 9.12 and Figure 9.16 present the results of this benchmark. The overhead results should be approximately the same for a given platform. We believe that the variations in some of the results are due to cache behavior differences.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	0.0075	0.0112	0.0067	0.0028
multi	NA	0.0112	0.0044	0.0028
ds	0.0075	0.0168	0.0044	0.0028
ps	0.0075	0.0112	0.0044	0.0028
pspr	0.0075	0.0113	0.0045	0.0028
psmp	NA	0.0169	0.0044	0.0056
pt	0.0076	0.0112	0.0067	0.0028

Table 9.12: Loop Overhead (in microseconds)

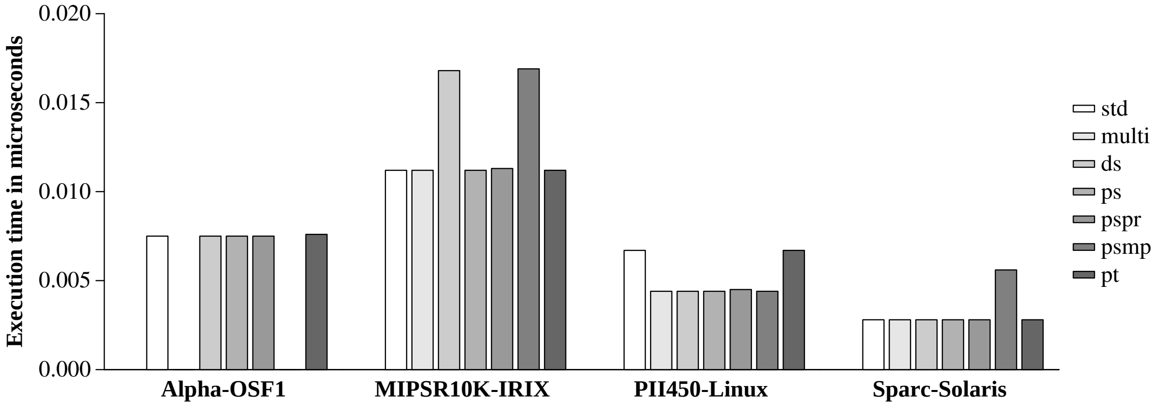

Figure 9.16: Loop Overhead

9.2.2.2 Local call

This benchmark measures the cost of making a local proc call with no arguments and no return value. Table 9.13 and Figure 9.17 present the results of this benchmark. In general, the uniprocessor targets perform better than the multiprocessor targets because they lack locking and thread identification overhead. For all the platforms with the exception of PII450-Linux, **ds** performs slightly better than **std**. However, **psmp** outperforms **multi** for all the platforms that support these targets. As expected, the **ps** target performs slightly worse than the **ds** target for all the platforms except for MIPSR10K-IRIX. The **pt** target exhibits the worst performance on all the platforms due to the high cost of locking and thread identification. Notice that the relative performance of the **pt** target varies from platform to platform. This indicates that the performance of native Pthreads implementations can vary greatly across platforms.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	0.1302	0.3139	0.0684	0.1777
multi	NA	0.3202	0.0941	0.2974
ds	0.1281	0.3070	0.0727	0.1002
ps	0.1306	0.2520	0.0794	0.1295
pspr	0.1289	0.3030	0.0763	0.1013
psmp	NA	0.3189	0.0935	0.2435
pt	0.2213	0.3858	0.2268	0.4732

Table 9.13: Local Call (in microseconds)

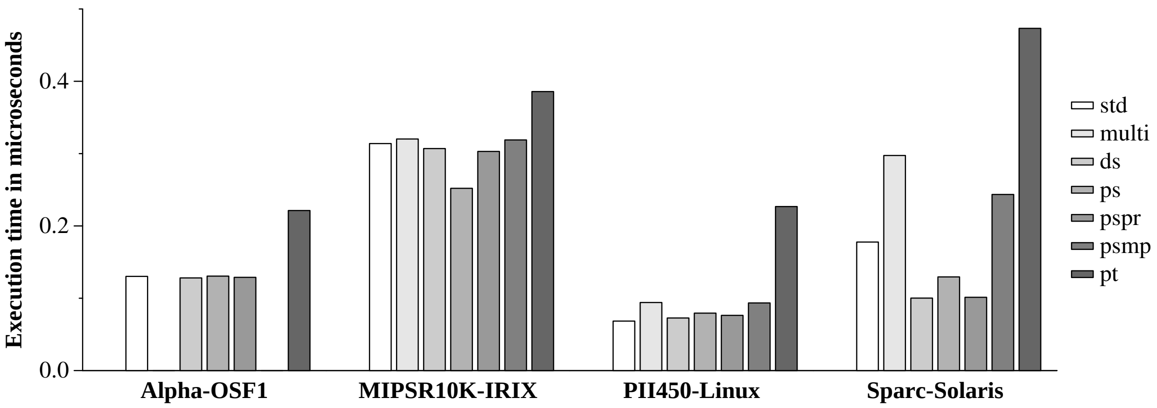

Figure 9.17: Local Call

9.2.2.3 Inter-resource Call – No New Process

This benchmark measures the cost of making an inter-resource call that does not create a new process to service the call. The test procedure has no arguments and no return value. Table 9.14 and Figure 9.18 present the results of this benchmark. For each target platform, **ds** performs better than **std** and **psmp** performs better than **multi**. Again, for each platform except MIPSR10K-IRIX, **ds** performs better than **ps**. Also, **pt** performs considerably worse than all the other targets. Notice that **pspr** usually performs worse than **ds** and **ds** due to the preemption overhead. This will generally hold for all the benchmarks.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	1.2927	2.3449	1.2000	1.5072
multi	NA	5.2614	2.0560	3.0024
ds	1.1294	2.2114	1.1970	1.4188
ps	1.2498	2.1061	1.2790	1.5560
pspr	1.6127	2.8110	1.3060	1.7733
psmp	NA	4.7965	1.7870	2.4817
pt	3.2734	7.4989	3.4270	4.0249

Table 9.14: Inter-resource Call – No New Process (in microseconds)

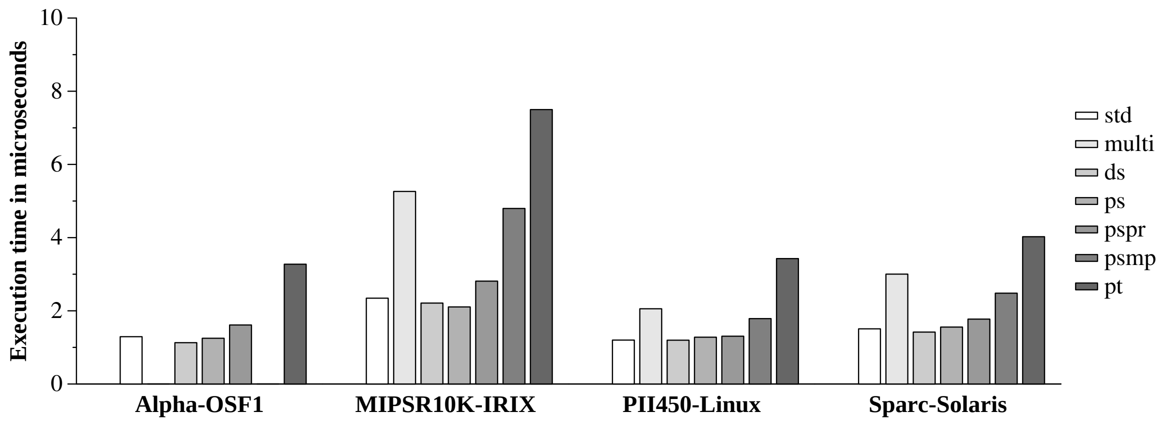

Figure 9.18: Inter-resource Call – No New Process

9.2.2.4 Inter-resource Call – New Process

This benchmark is similar to the previous benchmark, except that a new process is created to service the inter-resource call. Table 9.15 and Figure 9.19 present the results of this benchmark. This benchmark reveals larger discrepancies in performance among the various targets; in particular, the multiprocessor targets, **multi** and **psmp**, perform considerably worse than the uniprocessor targets. The **pt** target performs significantly worse than the others, exhibiting the high cost of thread creation in Pthreads. Another interesting result is that the **std** target has the best performance for all the platforms. However, the **psmp** target performs significantly better than **multi** on PII450-Linux and Sparc-Solaris. This performance difference is largely due to the benefit of using a preswitch-style context switch to avoid two context switches on multiprocessors. Locking and thread identification also contribute to the performance differences.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	3.7745	7.3472	2.8800	6.5669
multi	NA	22.7557	11.2850	24.5945
ds	4.4411	NA	3.5560	7.2814
ps	4.6338	NA	3.3500	9.2948
pspr	6.0032	NA	3.9260	10.3801
psmp	NA	NA	7.0990	13.1820
pt	80.3100	183.9500	204.8000	192.9800

Table 9.15: Inter-resource Call – New Process (in microseconds)

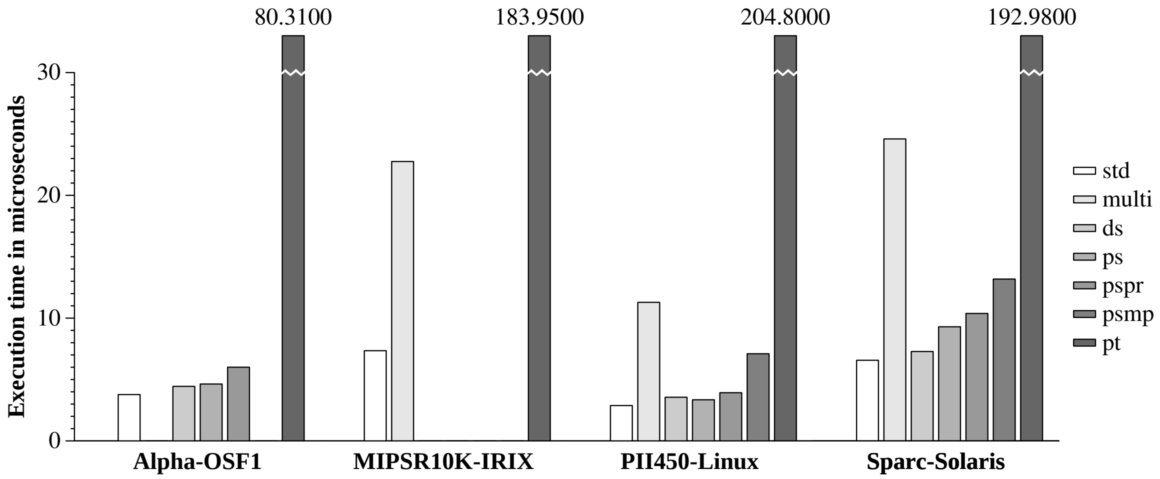

Figure 9.19: Inter-resource Call – New Process

9.2.2.5 Process Create/Destroy

This benchmark measures the cost of process creation and destruction. Table 9.16 and Figure 9.20 present the results of this benchmark. Similar to the previous benchmark, the performance of **multi** on the Sparc-Solaris platform is relatively much worse than **multi** on the other platforms. This is due to SR's use of two context switches and the fact that a context switch on the Sparc requires the register windows to be flushed via a system call.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	3.5372	6.6919	2.3880	6.4032
multi	NA	18.1938	10.1680	27.6444
ds	4.4066	7.7162	2.8370	10.7650
ps	4.5936	7.5633	3.3450	7.8205
pspr	5.7131	9.5309	4.0220	8.3775
psmp	NA	17.7937	6.3600	11.9601
pt	68.4900	162.4700	200.0000	177.2700

Table 9.16: Process Create/Destroy (in microseconds)

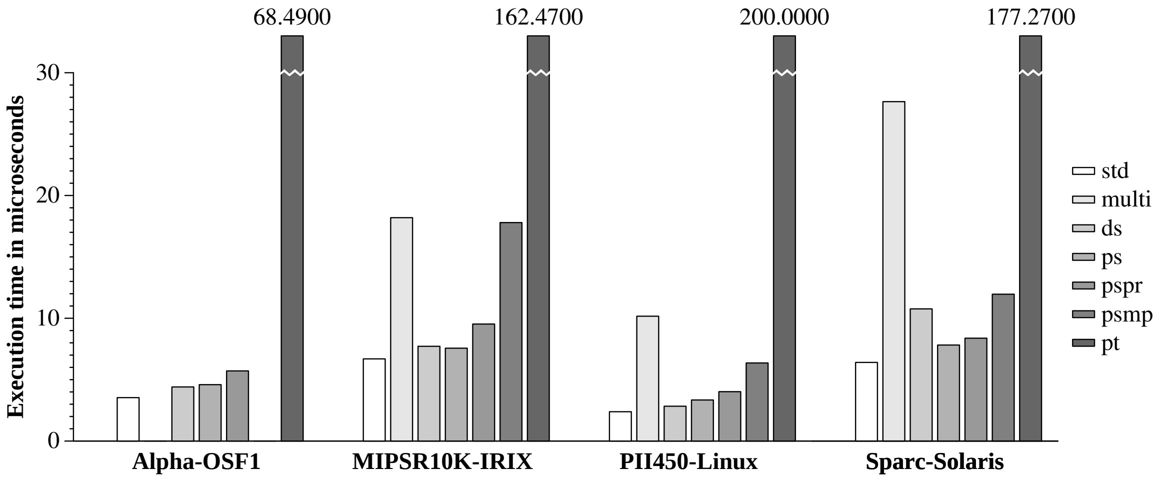

Figure 9.20: Process Create/Destroy

9.2.2.6 Semaphore Pair

This benchmark measures the cost of a semaphore **V** operation followed by a non-blocking **P** operation. It is designed to measure the cost of synchronization without blocking or context switching. Table 9.17 and Figure 9.21 present the results of this benchmark, which are similar to the results from the local call benchmark (see Section 9.2.2.2). Most of the differences in performance are due the differences in the cost of locking and thread identification. Again notice that the **std** target has the best absolute performance and that **psmp** outperforms **multi** due to faster locking and faster thread identification. However, in the case of PII450-Linux, the same locking and thread identification methods are used for both **psmp** and **multi**. The performance difference is due to a more efficient semaphore implementation afforded by Mezcla.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	0.1885	0.5659	0.1479	0.2278
multi	NA	2.6520	1.2544	1.9222
ds	0.1985	0.6441	0.1680	0.1969
ps	0.2007	0.6260	0.1666	0.1735
pspr	0.2798	0.7919	0.1943	0.3010
psmp	NA	1.5558	0.4468	0.6468
pt	1.0206	2.0840	0.9889	1.0771

Table 9.17: Semaphore Pair (in microseconds)

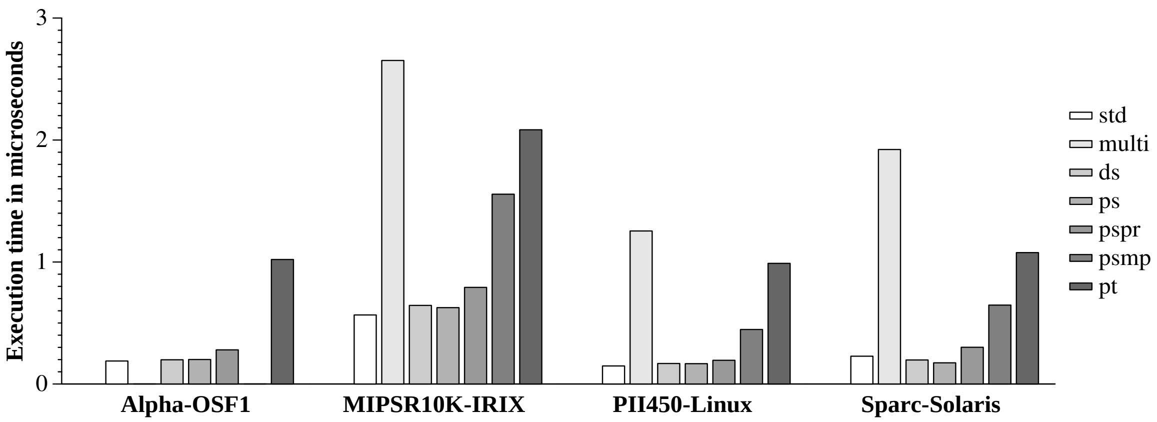

Figure 9.21: Semaphore Pair

9.2.2.7 Semaphore Requiring Context Switch

This benchmark measures the cost of semaphore synchronization with blocking and context switching using two processes and two semaphores. The two processes are forced to ping-pong between each other. Table 9.18 and Figure 9.22 present the results of this benchmark. For this benchmark, **ds** outperforms **std** and **psmp** outperforms **multi**. The **multi** execution time for the Sparc-Solaris target is particularly bad because each ping-pong requires four context switches.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	1.6460	4.1454	0.9782	4.7084
multi	NA	11.3144	5.8792	20.0252
ds	1.4938	3.8330	1.0314	4.3028
ps	1.6470	3.9516	1.0868	3.7856
pspr	2.2576	4.8442	1.2344	4.3352
psmp	NA	8.5836	2.7278	6.3084
pt	28.5840	108.9280	40.3000	59.5680

Table 9.18: Semaphore Requiring Context Switch (in microseconds)

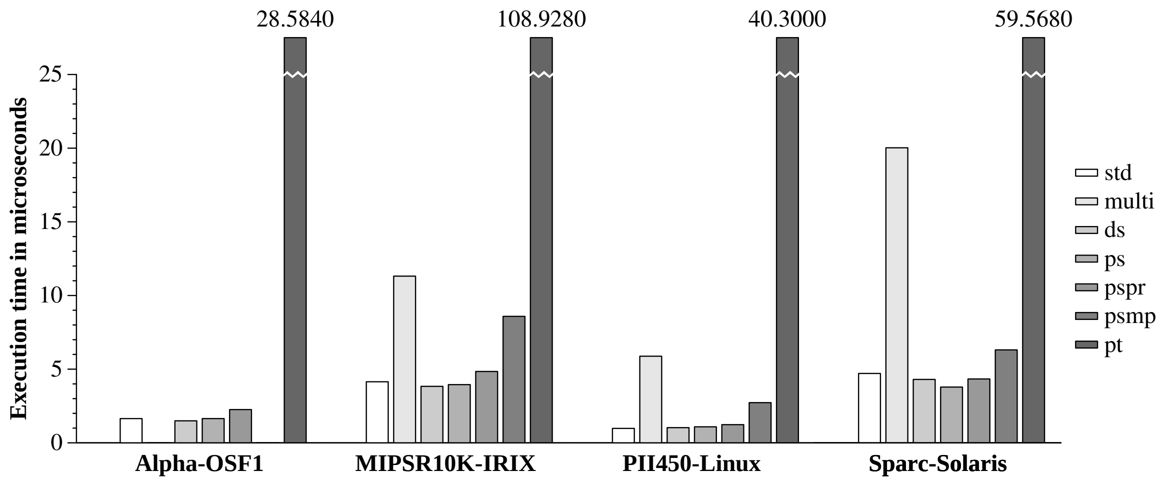

Figure 9.22: Semaphore Requiring Context Switch

9.2.2.8 Asynchronous Send/Receive

The asynchronous send/receive benchmark measures the cost of message passing without blocking. Table 9.19 and Figure 9.23 present the results of this benchmark. The results are similar to the semaphore pair benchmark (see Section 9.2.2.6). However, this benchmark exercises the invocation handling code in the SR run-time system. Recall that we made some significant modifications to SR invocation handling to use the Mezcla primitives. These modifications account for the better performance of **psmp** compared to **multi** on PII450-Linux and Sparc-Solaris. The other Mezcla targets perform similarly to the original targets.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	1.9442	3.3354	1.5030	2.1003
multi	NA	8.8416	3.1962	6.8083
ds	1.9512	3.0368	1.8056	1.7738
ps	1.8984	2.9582	1.8275	1.7677
pspr	2.5197	4.2335	1.7482	2.4073
psmp	NA	8.8868	2.9440	4.5014
pt	6.1522	13.1566	6.5336	6.7621

Table 9.19: Asynchronous Send/Receive (in microseconds)

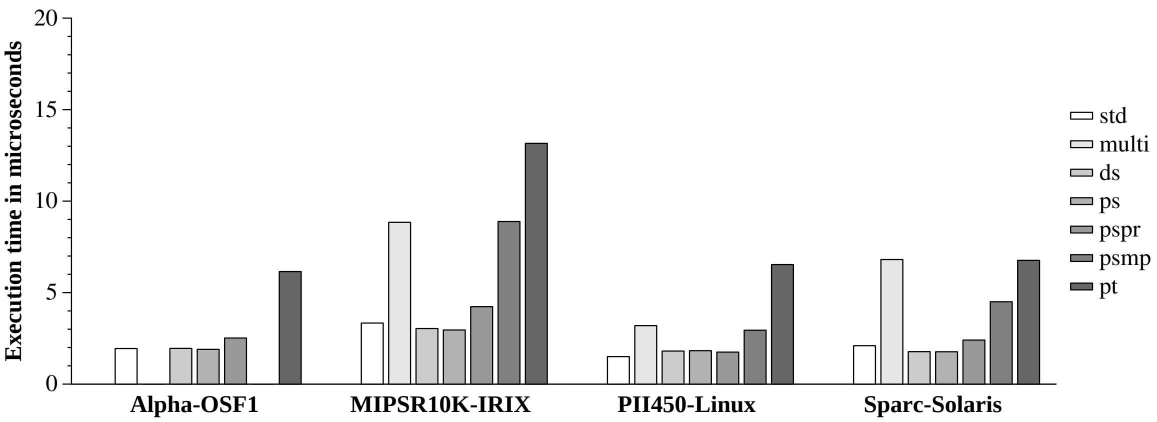

Figure 9.23: Asynchronous Send/Receive

9.2.2.9 Message Passing Requiring Context Switch

This benchmark measures the cost of message passing with blocking by having two processes ping-pong empty message between each other. Table 9.20 and Figure 9.24 present the results. For this benchmark, **ds** performs similarly to **std**, but **psmp** performs better than **multi**. As usual, the **pt** performs quite poorly. Most of the MIPSR10K-IRIX benchmarks fail. Even the **multi** target from the original SR implementation fails. (We did not track down the cause of these failures due to time constraints.)

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	5.7140	10.1200	4.3160	9.2980
multi	NA	NA	14.6800	29.8300
ds	6.1560	NA	4.5520	8.6420
ps	6.2180	NA	3.9660	7.5040
pspr	7.3380	NA	5.0220	10.3460
psmp	NA	NA	9.0060	18.9060
pt	44.5800	167.3200	71.4000	87.2600

Table 9.20: Message Passing Requiring Context Switch (in microseconds)

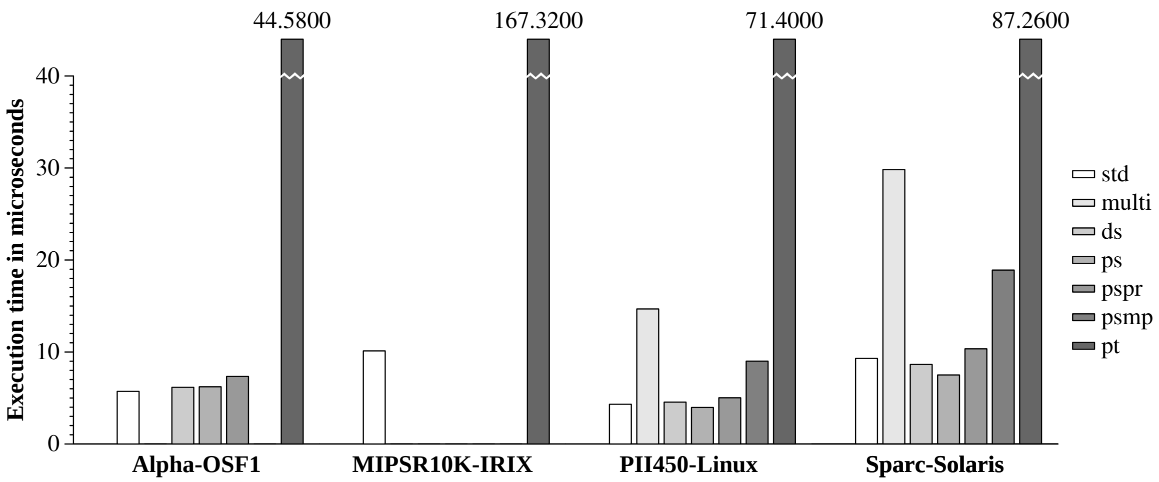

Figure 9.24: Message Passing Requiring Context Switch

9.2.2.10 Rendezvous

This benchmark measures the cost of rendezvous in SR. One process simply blocks on an operation with the **in** statement. A second process repeatedly invokes the operation synchronously. Table 9.21 and Figure 9.25 present the results of this benchmark. The overall pattern of results for this benchmark are similar to the previous benchmark (except for **pt** on MIPSR10K-IRIX, which performs poorly on the previous benchmark).

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	4.7920	8.6440	3.3100	7.6390
multi	NA	23.4450	14.5570	29.3490
ds	4.2570	NA	3.4830	7.7110
ps	4.4680	NA	3.9910	6.1710
pspr	5.8650	NA	3.7470	7.6180
psmp	NA	NA	7.5560	14.8200
pt	41.5500	37.4600	61.2700	34.9300

Table 9.21: Rendezvous (in microseconds)

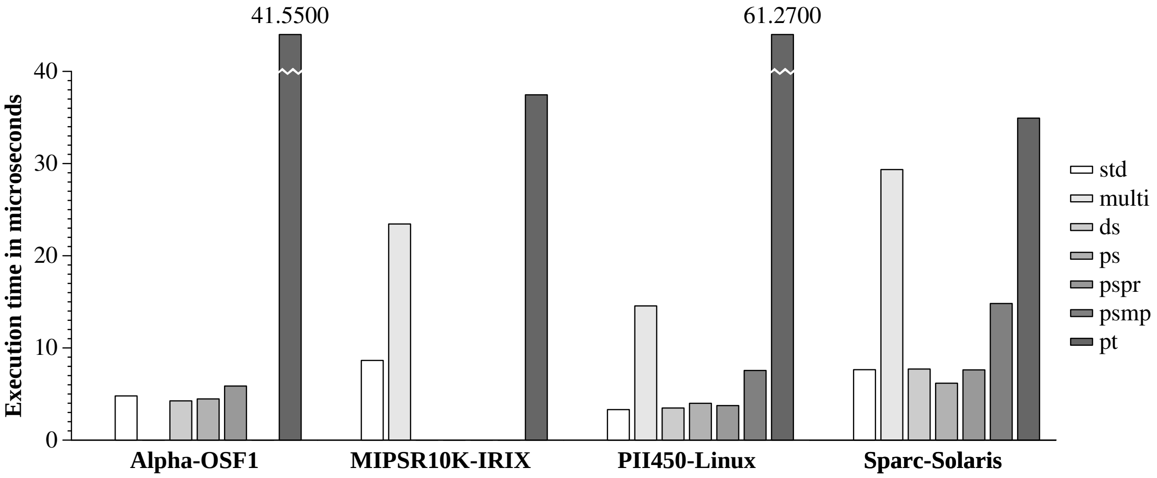

Figure 9.25: Rendezvous

9.2.2.11 Parallel Matrix Multiply

The final benchmark is a simple implementation of coarse-grained, parallel matrix multiply from [50]. Table 9.22 and Figure 9.26 present the results of this benchmark. They show that the performance of the individual synchronization mechanisms does not have a large impact on coarse-grain parallel applications. In general, the multi-processor targets exhibit speed-up; the speed up is nearly linear for MIPSR10K-IRIX and Sparc-Solaris, but not quite as good for PII450-Linux and Alpha-OSF1.

Target	Alpha-OSF1	MIPSR10K-IRIX	PII450-Linux	Sparc-Solaris
std	6.1360	7.9900	3.5770	12.4120
multi	NA	4.0540	2.2280	6.1410
ds	6.1340	7.8960	3.5820	12.2440
ps	6.1420	7.8940	3.5790	12.1090
pspr	6.3610	8.0100	3.5880	12.8620
psmp	NA	4.0500	2.1760	6.1240
pt	4.8840	4.0440	2.1950	6.7890

Table 9.22: Parallel Matrix Multiply (in seconds)

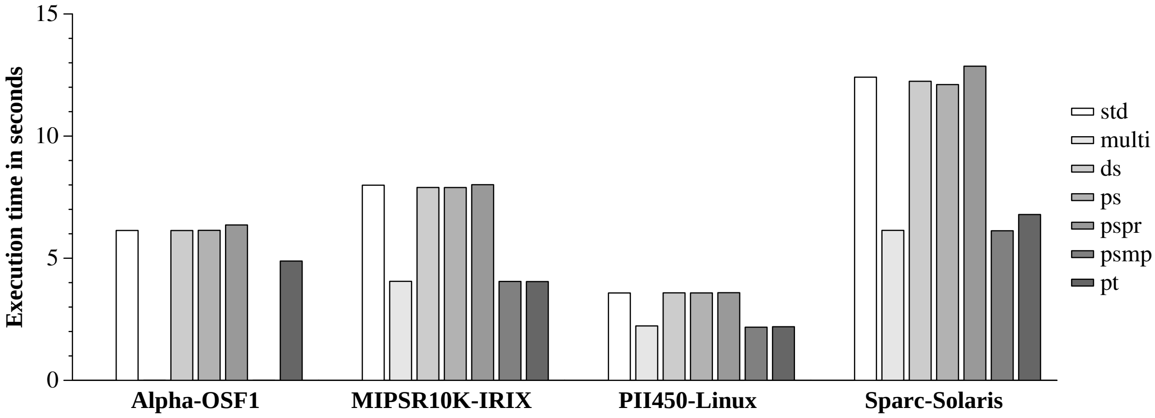

Figure 9.26: Parallel Matrix Multiply

9.2.3 Discussion

Our experience with implementing the SR run-time system using Mezcla was quite positive. In general, we observe that the Mezcla versions of SR perform as well as or better that the original implementations. For some benchmarks, the original uniprocessor version of SR exhibited the best absolute performance, showing that a true custom implementation can beat a Mezcla implementation in some cases. However, the overall performance difference is small, about 15% on average. The improved performance for multiprocessors and improved retargetability outweighs the very small benefit of a custom design. Also note that a Mezcla implementation of the SR run-time system designed from scratch could probably do as well or better than the original custom implementation. For example, we could design the run-time system so that thread functions do not rely on four arguments to improve the performance of process creation.

In addition to the favorable performance results, our coding experience was rather encouraging. The coding time was relatively short; just a few days to re-implement the SR run-time system using Mezcla. This ease of integration was largely due to Mezcla's ability to conform to the naming conventions of existing run-time systems. For example, Mezcla generates a custom TCB with the same name as the TCB implementation in the original source code (`proc_st`).

The Mezcla primitives provide a relatively clean approach to implementing unified run-time systems. For example, a benefit of using the Mezcla primitives and the Mezcla locking conventions is that a single run-time system can be compiled for uniprocessors, uniprocessors with preemptive scheduling, and multiprocessors.

The most significant shortcoming of the Mezcla implementation of SR is the lack of blocking I/O support. Therefore, the original implementation of the SR run-time system has more functionality than the Mezcla implementations. However, this added functionality does not account for the diminished performance results for original SR in the benchmarks. Specifically, blocking I/O support in the original SR run-time system is not on the critical path for the synchronization and process creation mechanisms. In addition, our benchmarks do do not rely on blocking I/O support.

9.3 Java

To further evaluate the Mezcla prototype, we ported the Kaffe Java virtual machine [116] to Mezcla. Unlike SR, Java supports only a limited number of concurrency mechanisms: thread creation, thread joining, and monitors (see Section 2.2.2 for a more complete description of concurrency in Java).

We chose to use the Kaffe Java virtual machine because it is freely available and currently does not support multiprocessors. One goal of porting Kaffe to Mezcla is to allow Kaffe to utilize Mezcla's multiprocessor targets.

9.3.1 Mezcla Implementation

The Mezcla implementation of Kaffe was straightforward; it simply re-implemented Kaffe's internal thread system, jthreads, using Mezcla. Unlike SR, Kaffe defines a platform-independent thread layer that is used for portability. Thus, new thread systems have to implement only the jthreads interface. The standard implementation of jthreads is a custom user-level thread system that uses `setjmp()` and `longjmp()` for context switching. The Mezcla client description defines a custom TCB for the jthreads layer. See Appendix E for a listing of the Mezcla client description file for Kaffe.

Unfortunately, the Mezcla implementation of Kaffe has two major shortcomings. First, like most Java virtual machines, Kaffe implements a garbage collector. Unfortunately, to see a consistent view of memory during garbage collection, no threads (except for the garbage collection thread) may be running. For uniprocessor implementations of Kaffe, it is relatively easy to ensure that no other threads are running. However, for multiprocessor implementations, it is much more complicated to "suspend" threads running on different processors. Currently Mezcla does not support the suspension of threads across different processors. Therefore, we can only run experiments without garbage collection. This restricts the type and affects the running time of the experiments.

The second shortcoming is that the Pthreads (**pt**) target is not fully functional in the Mezcla implementation of Kaffe. It turns out that incorporating Mezcla into Kaffe uncovered a rather fundamental design problem involving the Mezcla primitives and the **pt** target. The basic problem is that the jthreads interface provides mutex variables and condition variables. These are implemented using custom Mezcla primitives for blocking and unblocking. In Pthreads, a condition variable requires a corresponding mutex variable.

The associated mutex must be acquired before operating on the condition variable. More specifically, when `pthread_cond_wait()` is called, it requires a pointer to a condition variable and a pointer to a mutex variable. This function atomically blocks the calling thread and releases the mutex.

The problem is that the Mezcla implementation of the jthreads mutex and condition variables for the **pt** target does not directly map these functions to the Pthreads equivalents. Thus, a jthreads mutex variable differs from a Pthreads mutex variable. This means that a call to `jmutex_lock` will acquire a *logical* lock, but not a "real" Pthreads mutex lock (however, a Pthreads mutex lock is used to implement the logical lock). Thus, it is impossible to implement `jcondvar_wait()` such that it directly calls `pthread_cond_wait()` with a "real" locked Pthreads mutex variable. It is unclear how to alter Mezcla to deal with this problem. However, this problem does reinforce the fact that fully supporting the **pt** target is a difficult task.

9.3.2 Experiments

Due to the shortcomings mentioned above, we can run only a limited number of benchmarks. We have selected a small number of microbenchmarks to help predict how a full Mezcla implementation of Kaffe might perform. Kaffe is fully supported on only two of our four test platforms: PII450-Linux and Sparc-Solaris. Therefore, we present results for only these two platforms. In the following tables and graphs, **org** denotes the original Kaffe implementation.

9.3.2.1 Loop Overhead

The Java loop overhead benchmark is similar to the previous loop overhead benchmarks (see Sections 9.1.3.1 and 9.2.2.1); it measures the cost of running a loop in following benchmarks. Table 9.23 and Figure 9.27 present the results of the loop overhead test.

Target	PII450-Linux	Sparc-Solaris
org	0.0782	0.0755
ds	0.0734	0.0751
ps	0.0734	0.0753
pspr	0.0714	0.0755
psmp	0.0735	0.0751
pt	0.0733	0.0751

Table 9.23: Loop Overhead (in microseconds)

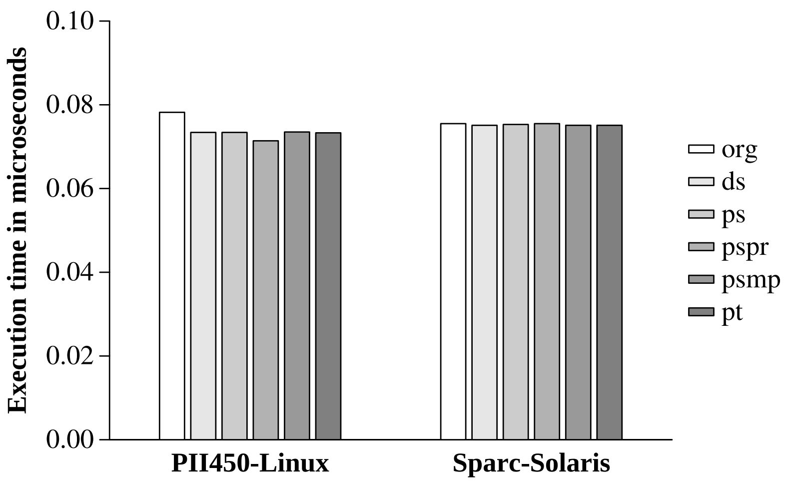

Figure 9.27: Loop Overhead

9.3.2.2 Method Invocation

The method invocation benchmark measures the cost of a simple method invocation on a Java object. The test method takes no arguments, but returns an integer. Table 9.24 and Figure 9.28 present the results of this benchmark. In general, all of the targets perform similarly, with the Mezcla target performing slightly better than the original Kaffe implementation. This is likely due to locking overhead. The original implementation of Kaffe supports preemption in a similar manner to the **pspr** target. Locks in the original implementation simply disable preemption.

Target	PII450-Linux	Sparc-Solaris
org	0.180	2.328
ds	0.150	2.278
ps	0.150	2.255
pspr	0.160	2.255
psmp	0.160	2.323
pt	0.220	2.477

Table 9.24: Method Invocation (in microseconds)

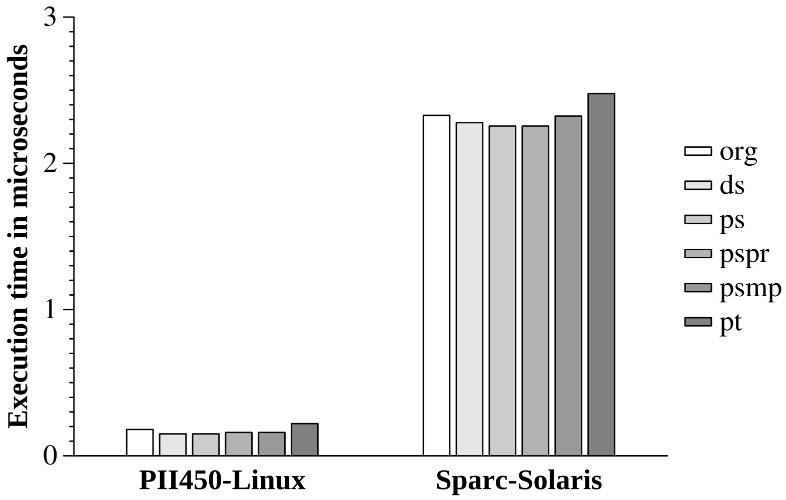

Figure 9.28: Method Invocation

9.3.2.3 Synchronized Method Invocation

The synchronized method invocation benchmark measures the cost of method invocation on a synchronized object. Table 9.25 and Figure 9.29 present these results. They show that the Mezcla uniprocessor targets perform better than the original Kaffe implementation and that the **psmp** target performs worse than **org**. However, **psmp** has greater functionality in that it can run Java threads on multiple processors, so the extra time is justified.

Target	PII450-Linux	Sparc-Solaris
org	0.580	3.040
ds	0.420	2.845
ps	0.430	2.654
pspr	0.450	2.849
psmp	0.950	3.832
pt	2.190	2.681

Table 9.25: Synchronized Method Invocation (in microseconds)

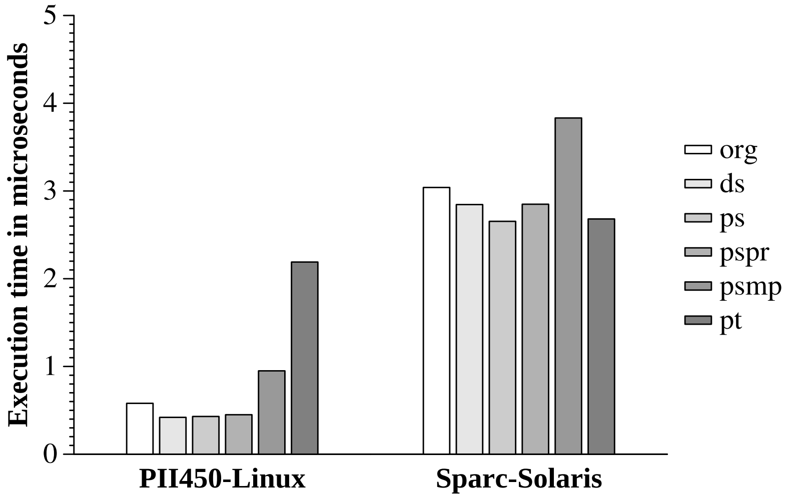

Figure 9.29: Synchronized Method Invocation

9.3.2.4 Thread Create/Join

The next benchmark measures the cost of thread creation and joining. Table 9.26 and Figure 9.30 present the results of the thread create/join benchmark. For this benchmark, all of the Mezcla targets, even the **psmp** target, perform significantly better than the original Kaffe implementation. This is largely due to the overhead in context allocation and initialization in the original implementation.

Target	PII450-Linux	Sparc-Solaris
org	345.0	753.2
ds	182.0	612.6
ps	182.0	611.4
pspr	184.0	616.1
psmp	205.0	637.1
pt	NA	NA

Table 9.26: Thread Create/Join (in microseconds)

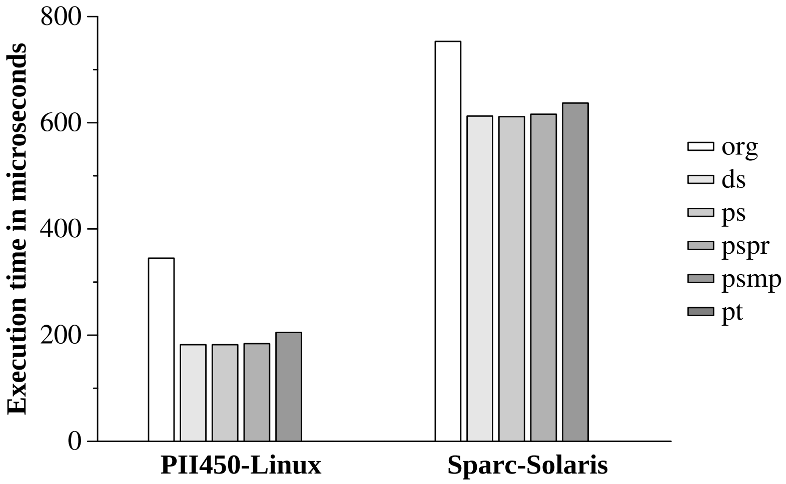

Figure 9.30: Thread Create/Join

9.3.2.5 Parallel Counting

This benchmark is a trivial parallel program that demonstrates that the **psmp** version of Kaffe can run Java threads on a multiprocessor. The benchmark simply starts two threads that count to a specified number (10^8 in for these experiments). In addition, each thread computes a running sum of each number counted. Table 9.27 and Figure 9.31 present the results. The **psmp** target exhibits near linear speed-up on both platforms.

Target	PII450-Linux	Sparc-Solaris
org	16.52	19.96
ds	15.61	19.86
ps	15.62	19.86
pspr	16.52	19.92
psmp	8.03	10.98

Table 9.27: Parallel Counting (in seconds)

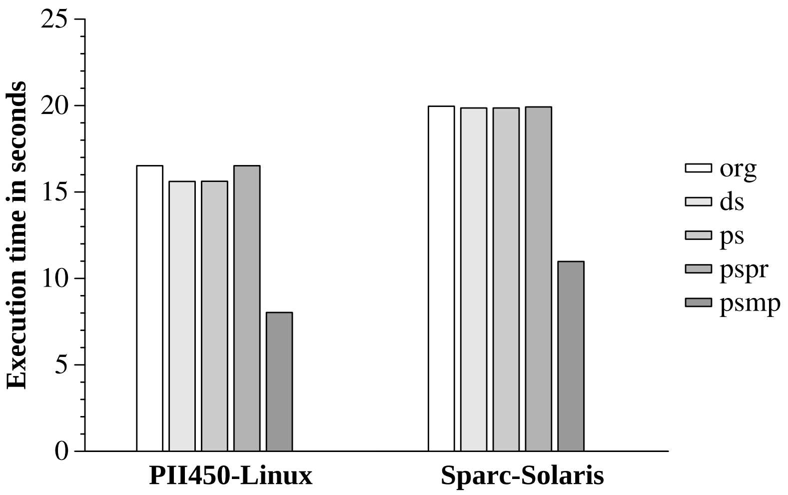

Figure 9.31: Parallel Counting

9.3.3 Discussion

In general, for our limited number of microbenchmarks, the Mezcla implementation performed similarly to or better than the original Kaffe implementation. Our initial data suggests that a full Mezcla implementation of Kaffe would improve the performance of Java applications that create and synchronize many threads.

Unfortunately, attempting to implement Kaffe with Mezcla revealed two major shortcomings of the current Mezcla prototype. First, languages with garbage collection need support for suspending all running threads (even if they are running on different processors). Mezcla does not currently support thread suspension. Second, the Mezcla implementation of Kaffe uncovered restrictions with the **pt** target. It is not clear if this is a problem or simply a restriction in using the Mezcla primitives. Further work is needed on the Mezcla prototype to address the shortcomings and to facilitate a full implementation of Kaffe.

9.4 Summary

This section demonstrated how to use the Mezcla thread framework for building custom thread systems. For each thread system, we provided detailed performance results for several benchmarks. Overall, the Mezcla-based implementations of SR and Java performed about as well or better than the original implementations. We observed performance improvements from 10% to 200% depending on the target. The best improvement came from better multiprocessor support in the **psmp** target compared to the original MultiSR implementation.

The Mezcla framework proved relatively easy to use. By using the Mezcla primitives, a language run-time system can be retargeted for a variety of target types. However, Mezcla only helps with retargeting thread operations, not other system-specific issues. The Mezcla-based implementation of Kaffe revealed some deficiencies in the Mezcla prototype: the inability to suspend threads for garbage collection and a restriction with the **pt** target. In addition, we ran into several portability problems across the different platforms that were unrelated to threads. Therefore, Mezcla is not a complete portability solution.

The next chapter make some concluding remarks and discusses directions for future work partially based on the experience presented in this chapter.

Chapter 10

Conclusions

This chapter offers a reflection on the work presented in this dissertation. It summarizes the research results and experience. Then, it looks at both the applicability and the limitations of the Mezcla thread framework. It also offers recommendations for operating system developers and programming language implementors. This chapter finishes with a discussion of future work.

10.1 Summary of Results and Experience

One of the main results of the work presented in this dissertation is that it is possible to generate efficient, specialized thread systems that blend the requirements of a language run-time system or thread library with the functionality of a particular operating system and hardware architecture. The key challenge for achieving thread generation is the separation of client concerns from target concerns. The Mezcla prototype described in Chapter 8 introduced an approach for achieving this separation by using a simple client description format and target templates written in annotated C code. A thread generation tool combines client and target descriptions to create a *custom* thread system.

To evaluate the Mezcla prototype, we developed and measured the performance of Mezcla thread systems for three clients: SemThreads, the SR programming language, and a Java virtual machine. The results indicate that Mezcla-based implementations can perform as well as or better than the original language implementations. This is especially true for multiprocessor targets, in which Mezcla can optimize context switching and synchronization costs. We observed performance improvements from 10% to 200% depending on the benchmark, target, and platform.

Developing a thread system with Mezcla is relatively easy and gives the programmer complete flexibility in defining basic thread mechanisms. One might believe that it is easier to build a language run-time system on top of an existing thread system such as Pthreads. Although Pthreads does offer a fairly simple programming model, it is often the case (see Chapters 4 and 5) that a run-time system developer must "code around" the standard interfaces provided by Pthreads to get desired functionality. In addition, as demonstrated in Chapter 9, most Pthreads implementations have poor performance for basic thread operations. Thread systems for libraries and run-time systems that do not easily map to Pthreads and demand high performance are good candidates for using the Mezcla thread framework. Mezcla-based thread systems are tailored to meet the needs of a particular run-time system; they support the smallest amount of functionality required by the client. Therefore, Mezcla-based thread primitives have minimal critical paths.

10.2 Applicability of the Framework

The Mezcla thread framework is designed to be used by developers of run-time systems for concurrent programming languages. Based on experience using the Mezcla thread framework for SR and Java (see Chapter 9), it seems likely that the framework can also be applied to other concurrent programming languages. Our experiments focused on languages with explicit parallelism. However, since the Mezcla framework is a rather low-level thread generation tool, it could also be used in the run-time systems for languages with implicit parallelism such as functional languages and languages with parallelizing compilers. Implementations of these languages often rely on run-time systems with thread support. Mezcla can be used to help develop specialized thread primitives for these parallel run-time systems.

While the original impetus for the framework was to create efficient run-time systems, we believe that the framework can be applied to other types of software. In particular, the SemThreads experience presented in Section 9.1 showed that the framework can be used to create custom thread libraries to be used by applications. In addition, it is possible for application writers to directly use the Mezcla thread framework for developing application-specific thread systems. This could be especially useful if the application needs to support a large amount of thread specific data. For example, large-scale server programs (such as web servers) and multi-threaded simulation software are both classes of applications that often use a significant amount of thread-specific data. Using the framework, the application developer can create a Mezcla client specification with application-specific, per thread data in the thread control block description.

10.3 Limitations of the Framework

The prototype of the Mezcla framework presented in Chapter 8 realized the design goals presented in Section 6.1.1. These goals consist of four key framework properties: support for creating *specialized threads systems*, the ability to generate thread primitives with *minimal critical paths*, thread system *portability*, and thread system *retargetability*. However, the prototype does have some limitations.

First, the client description language, while simple, is very low-level. A programmer must understand how the client-specified C code will be used in order to write proper descriptions. A better approach might be to use a higher-level description language that generates code for both the client and the target, similar to the way an IDL (interface definition language) compiler generates code from a definition file for implementing remote procedure calls.

Second, the target templates are somewhat complicated because they are essentially parameterized C programs. As a result, a change to the overall structure of Mezcla generated code requires changes to all the supported targets. Currently, the prototype supports five target types. (Recall that a target type is a general approach to implementing a thread system that is not tied to a specific platform; a sample target type is the preswitch multiprocessor, **psmp**.) In addition, writing a new target type is somewhat tedious. Currently, the best way to write a new template is to have a good understanding of how other targets are implemented. Ultimately, it would be best to have a more abstract description of target types.

Third, for Mezcla-based thread systems to be truly portable, the Mezcla framework must support a large number of target platforms. The prototype supports only four target platforms. (Recall that a target platform is a specific operating system and processor architecture pair, such as Linux/x86 or Solaris/Sparc.) To support a new platform, the platform-specific library code for context switching and multiprocessor operation must be written for the platform. This situation is similar to programming language compilers.

In order for a language to be portable, it must have a compiler on many targets. Writing platform-specific support libraries is similar to, but not as complex as, writing a new compiler back end.

Finally, the Mezcla prototype does not currently support input/output or thread-related timing functions (e.g., putting a thread to sleep for a specified amount of time). Currently, the client must provide input/output support. This is a difficult problem, because how input/output is handled can vary greatly from platform to platform. Further research is needed to determine how to separate client input/output needs from platform-specific input/output interfaces.

10.4 Recommendations for Operating System Developers

The current level of operating system support for writing user-level thread systems is rather limited. A thread-system developer must write or obtain processor-specific context switching code as well as multiprocessor locking and identification routines. In addition, a developer must use several tricks to implement preemptive user-level thread scheduling and to schedule threads on specific processors. Ideally, an operating system would provide a library of mechanisms for building user-level thread systems. The following thread building mechanisms would be helpful:

- **Context switching code** An operating system should provide various forms of processor-specific context switching code. Most kernels require context switching code internally; this code could be put into a publicly available library or header file so that it is accessible to programmers. The C library routines `setjmp/longjmp` provide a limited form of context switching, but to implement threads with these functions, a programmer still has to have some processor-specific knowledge. A library of context switching code should provide support for different types of switching, such as a switch mechanism that saves just the integer registers or both integer registers and floating point registers on a context switch. Both the direct switch and preswitch methods should be supported.

- **Multiprocessor locking** Current operating system support for multiprocessor locking varies greatly. Some operating systems, such as Linux, provide no support; the programmer must implement multiprocessor-safe locks directly using processor instructions. Other operating systems, such as IRIX, provide a low-level interface to multiprocessor-safe locks that coordinate with the kernel scheduler. Finally, some operating systems, such as Solaris, provide only high-level interfaces to multiprocessor-safe locks. A set of thread building mechanisms should include two classes of low-level processor-specific locks: those that do not directly coordinate with the kernel and those that coordinate with the kernel by blocking after a programmable spin duration.

- **Fast thread identification** A key element in building a user-level thread system on a multiprocessor is the ability to quickly identify which thread is currently running. Often, this can be achieved with a system call, like `getpid()`. Unfortunately, system calls can be relatively slow. A faster method is to use a processor-specific register that can identify the current execution context. An operating system should provide a machine-independent interface to thread identification in a publicly available header file.

- **Preemptive scheduling in user space** The most common way to support preemptive user-level threads is to set a timer signal. However, the signal interface was not originally designed to be used for thread preemption, so the programmer must either use slow system calls to prevent a timer signal, or the signal handler must be programmed to avoid preemption when a user-level thread is in a critical section. Neither solution is ideal. The former is correct, but inefficient; the latter is efficient, but it might introduce long scheduling delays. A better kernel interface is needed to support preemption in user-level threads.

- **Kernel cooperation** Most operating systems do not adequately support user-level thread systems ([7] and [82]). The main problem is that kernel-level schedulers do not cooperate with user-level schedulers. Therefore, the kernel-level scheduler can make a decision that will adversely affect a multi-threaded, user-level application. The Scheduler Activation approach [7] offers a solution to this problem. Both Solaris and Digital UNIX have support for Scheduler Activations, but the interfaces are undocumented and not made available to application programmers. Operating systems should provide documented mechanisms that allow kernel schedulers to better cooperate with user-level schedulers.

All of the mechanisms listed above could be part of a single standard, perhaps as a new POSIX standard, for building user-level thread systems. A standard set of thread building mechanisms would make it much easier to implement the Mezcla thread framework and to make it more portable. In addition, the Mezcla thread framework would be a good way for language implementors to take advantage of advanced operating system support for user-level threads without having to directly program the mechanisms. The programmer would be distanced from the platform-specific features by using the client description language. Ultimately, something like the Mezcla thread generation tool could become as common as C compilers.

10.5 Recommendations for Language Implementors

A programming language implementor is faced with many design choices when developing a concurrent run-time system. The mainstream approaches to structuring run-time systems presented in Section 2.3.4 do not afford a good compromise between performance and portability for languages that rely on efficient, language-specific thread semantics. This key observation motivated the development of the Mezcla thread framework. At this point, the Mezcla prototype is very much a "research" class system. While it could be used to developed efficient and portable run-time systems, it will require further development to improve its robustness and to increase the number of supported targets.

Our work in developing the Mezcla prototype and incorporating it into the run-time systems of concurrent programming languages has provided us with some useful insight into both run-time system and language design. We explored several key design areas and offer recommendations in each:

- **Locking Discipline** A concurrent run-time system requires a locking strategy to ensure mutual exclusion to run-time system data structures. Two general approaches to locking are coarse-grain and fine-grain. Typically, in the coarse-grain approach, a single lock is used to protect all run-time system data. In the fine-grain approach, different data elements have different locks; therefore, a locking hierarchy must be defined for concurrent access to multiple data elements. The fine-grain approach has the potential to alleviate contention for a lock protecting multiple, unrelated data elements. However, this approach can lead to complicated locking code. We believe an implementor should start with the coarse-grain approach, then determine if a fine-grain approach is necessary. Fine-grain locks can be added to the run-time system as necessary.

- **Language-level Synchronization** Many concurrent languages, such as SR and Java, provide *only* language-level support for synchronization. Language-level synchronization mechanisms often make it easier to write concurrent programs and, because they are defined by the language, they will work on any platform supported by the language implementation. However, for some systems applications it might be desirable to have access to low-level locks in a high-level language. The language-level mechanisms, while useful, often have high overhead for performing simple mutual exclusion. We suggest that language designers provide a more efficient way for programmers to access low-level locks for fast mutual exclusion. Such access could potentially allow a programmer to implement custom synchronization mechanisms not included in a language's definition.

- **Language-level Input/Output** Incorporating input/output access in a high-level language is a notorious problem. The main question is: Should input/output be a part of the language definition or should input/output be made available as a library? The former approach leads to better portability, but it can limit the available number and types of input/output operations. Also, it cannot easily accommodate new or specialized forms of input/output operations. The latter approach is perhaps more flexible for the programmer, but it makes a concurrent run-time system more complicated. This is because input/output can often affect thread scheduling, so a run-time system needs to be aware of the way input/output operations are to be used. How can there be agreement between arbitrary input/output libraries and the core run-time system? The best one can do at this time is to carefully design a run-time system so that the interdependencies between the language run-time system and the input/output libraries are well-defined.

10.6 Future Work

The work presented in this dissertation can be extended in several directions. In addition, this dissertation has opened up several new research areas. Possible Mezcla extensions and Mezcla-related project ideas include:

- **Prototype Extensions** Perhaps the most obvious step is to extend the current Mezcla prototype. Possible extensions include adding target types and target platforms, making a richer client description language, simplifying the target description language, providing better input/output support, and including more sophisticated support for multiprocessor scheduling algorithms.

- **A Compiler Approach to Building Thread Systems** As proposed in Chapter 6, it might be possible to develop a domain specific language for specifying thread systems. A compiler would read thread description programs, translate them into an intermediate representation, perform platform-independent optimizations, then generate a platform-dependent thread system. Instead of using annotated C code, platform descriptions could be more concise; they could provide just the essential features of a particular target. The code generation phase could perform platform-dependent optimizations, such as cache conscious data alignment.

- **Operating System Integration** As proposed in Section 10.4, Mezcla could generate better user-level thread systems given better operating system support for threads. One approach is to develop better operating system thread building mechanisms for some of the free operating systems, such as Linux, NetBSD, or FreeBSD. Mezcla target descriptions could be developed in concert with new kernel mechanisms.

- **Locking Framework** Perhaps one of the most difficult challenges for a thread system implementor is designing a robust locking strategy. While Mezcla does not enforce any particular strategy, it does require the programmer to use low-level locks and to adhere to a locking convention. Since locking is such a difficult issue, better tools are needed for implementing a locking strategy inside run-time systems. For example, in some cases it might be desirable to use a single lock to provide mutual exclusion inside a run-time system, while in other cases it might be better to use several, fine-grain locks. It would be nice to have a framework that allows compile-time selection of different locking granularities as well as tools to analyze the performance of different applications using different approaches. A more sophisticated locking system might allow for dynamic selection of locking methods.

- **Wait-free Synchronization Framework** Similar to the locking framework described above, it would be advantageous to have a framework for implementing different forms of wait-free synchronization. Currently, wait-free synchronization is not commonly used, most likely due to the complexity in writing algorithms and data structures based on wait-free synchronization. It might be possible to incorporate wait-free synchronization techniques into the Mezcla thread framework. The basic idea would be to hide the details from the thread system implementor via the client description language. This approach might be more feasible using the compiler approach to building thread system mention previously.

- **Mezcla Network Protocol Framework** Another research direction is to apply the same approach taken in the Mezcla thread framework to network protocols. That is, the Mezcla network protocol framework would allow a programmer to easily generate run-time specific network protocols using a simple specification file. Ideally, the generated protocols would perform better than using standard, layered protocols such as TCP. Such a framework could be useful for developing portable, high-performance run-time systems for clusters of workstations. This approach is different from the *x*-kernel in that the programmer does not have to explicitly program protocols; rather, they will be generated automatically from simple specification files.

Appendix A

Mezcla Prototype Source Code

A.1 Thread Generation

A.1.1 tgen

```
#!@PERL_PATH@ -w

# Add mtf library to perl search path
$mtf_lib_dir="@prefix@/lib";
push (@INC, $mtf_lib_dir);

process_args();

# Use "require" to allow process_args() to set @INC first

require 'mtf_template_parser.pm';


#-------------------------------------------------------------------------
# Translate the template file to an intermediate Perl file. Skip
# this step if the intermediate Perl file exists, and is newer than
# the template.
#-------------------------------------------------------------------------

my $compile_template = 0;
if (-e $inter_file) {
    if ((-M $inter_file) >= (-M $template_path)) {
        $compile_template = 1;
    }
} else {
    $compile_template = 1;
}
if ($compile_template) {
    if (mtf_template_parser->parse ($template_path, $inter_file) == 0) {
        print ("Translated $template_path to\n") if $verbose;
print ("           $inter_file\n") if $verbose;
    } else {
        die "Could not parse template file - exiting\n";
    }
}
```

```
#------------------------------------------------------------------------
# Parse the input specification file
#------------------------------------------------------------------------

require "${spec_parser}.pm"; $spec_parser->import;
$ROOT = $spec_parser->parse($spec_file);
print ("Parsed $spec_file\n") if $verbose;
$ROOT->print() if $debugging;

#------------------------------------------------------------------------
# Eval the intermediate Perl file and terminate
#------------------------------------------------------------------------

require $inter_file;
die "$@\n" if $@;
exit(0);

#------------------------------------------------------------------------
# process_args - process the command line arguments
#------------------------------------------------------------------------

sub process_args {
    $verbose = 1; $debugging = 0;
    $template_dir = $ENV{"MTF_TEMPLATE_DIR"} || $mtf_lib_dir || "";
    $template_file = "mtf_ds.tpl"; # default
    $spec_parser = "mtf_client_parser";         # default
    if (exists ($ENV{"MTF_OPTIONS"})) {
        print  "Using command line options from \"MTF_OPTIONS\" \n";
        @ARGV = split (/\s/,$ENV{"MTF_OPTIONS"});
    }
    while (@ARGV) {
        $a = shift @ARGV;
        if ($a eq "-h") {
            Usage();
        } elsif ($a eq "-s") {
            $spec_parser = shift @ARGV;
        } elsif ($a eq "-d") {
            $debugging = 1;
        } elsif ($a eq "-q") {
            $verbose = 0;
        } elsif ($a =~ /^-[Tt]$/) {
            $template_file = shift @ARGV ;
        } elsif ($a eq "-ti") {
            $inter_file = shift @ARGV ;
        } elsif ($a eq "-D") {
            my $code = shift @ARGV;    #-D foo=20 becomes ...
            eval("\$$code");           #eval ("$foo=20")
            die "Error in -D $code:\n$@\n" if $@;
        } else {
            $spec_file = $a;
        }
    }

    $template_found = 0;

    $template_path = $template_file;

    while (1) {
        if ($template_path && (-e $template_path)) {
            if (! $inter_file) {
#                $inter_file = "$template_file.pl";
                $template_found = 1;
                last;
```

```
            }
        }
print "template_path = $template_path\n" if $debugging;
        if ($template_path !~ m'/') {
            $template_path = "$template_dir/$template_file";
        } else {
            last;
        }
    }
    $inter_file = "$template_file.pl";

    if (! $template_found) {
        print "Please specify a template file\n";
        Usage();
    }
    if ((! $spec_file) || (! -e $spec_file)) {
        print "Please specify a valid specification file\n";
        Usage();
    }
    if (exists $ENV{"MTF_LIBDIR"}) {
        push (@INC, split(/:/, $ENV{"MTF_LIBDIR"}));
    }
}

#--------------------------------------------------------------------------
sub Usage {
    print <<"_EOT_";

Usage: tgen <options> <specification file>
 where options are:
 -t <template file>            : Name of the template file.
                                 Default : $template_file
                                 Default template directory = ".", which
                                 can be modified with MTF_TEMPLATE_DIR
 -q                            : Quiet Mode
 -d                            : Set a debugging trace. This is NOT quiet!
 -s <specification parser>     : Parser module that can parse the input
                                 specification file
                                 Default : "mtf_client_parser"
[-ti <intermediate perl file>] : tgen translates the template file to
                               : perl code. Default : "<template>.pl"
-D var[=value]                 : Define variables on the command line

The command line can be specified in the envt. variable "MTF_OPTIONS".

The pathname to all mtf modules can be set in the environment variable
    "MTF_LIBDIR" (colon-separated);
_EOT_
    exit(1);
}
```

A.1.2 mtf_ast.pm

```
package mtf_ast;

#---------------------------------------------------------------------------
# This package is used to create a simple Abstract Syntax tree. Each node
# in the mtf_ast is an associative array and supports two kinds of properties,
# scalars and lists of scalars.
#---------------------------------------------------------------------------
```

```
use strict;
my $curr_level = 0;
my $indent     = "   ";

# Constructor
# e.g mtf_ast->new ("nodename")
# Stores the argument in a property called mtf_astNodeName whose sole purpose
# is to support print()
sub new {
    my ($pkg, $name) = @_;
    bless {'mtf_ast_node_name' => $name}, $pkg;
}

# Add a property to this object
# $mtf_ast_node->add_prop("name", "value");
sub add_prop {
    $_[0]->{$_[1]} = $_[2];
}

# Equivalent to add_prop, except the property name is associated
# with a list of values
# $class_mtf_ast_node->add_prop_list("attr_list", $attr_mtf_ast_node);
sub add_prop_list {
    my ($this, $prop_name, $node_ref) = @_;
    if (! exists $this->{$prop_name}) {
        $this->{$prop_name} = [];
    }
    push (@{$this->{$prop_name}}, $node_ref);
}

# Returns a list of all the property names of this object
sub get_props {
    my ($this) = $_[0];
    return keys %{$this};
}
sub get_prop_value {
    my ($this, $prop_name) = $_[0];
    return $this->{$prop_name};
}
my @saved_values_stack;
sub visit {
    no strict 'refs';
    my $this = shift;
    package main;
    my ($var, $val, $old_val, %saved_values);
    while (($var,$val) = each %{$this}) {
#print "var = $var  val = $val\n" if $main::debugging;
        if (defined ($old_val = $$var)) {
           $saved_values{$var} = $old_val;
        }
        $$var = $val;
    }
    push (@saved_values_stack, \%saved_values);
}

sub bye {
    my $rh_saved_values = pop(@saved_values_stack);
    no strict 'refs';
    package main;
    my ($var,$val);
    while (($var,$val) = each %$rh_saved_values) {
        $$var = $val;
    }
```

```
}

# Recursively prints the entire mtf_ast tree.
sub print {
    my $this = shift;
    my($curr_indent);
    my($i,$o,$prop);
    $curr_indent = $indent x $curr_level;
    print "${curr_indent}label: ", $this->{"mtf_ast_node_name"}, "\n";
    $curr_indent .= $indent ;
    ++$curr_level;

    foreach $prop (keys %$this) {
        next if ($prop eq "mtf_ast_node_name");
        $o = $this->{"$prop"};
        if (ref($o) eq "mtf_ast") {
            $o->print();
        } elsif (ref($o) eq "ARRAY") {
    if (ref(@{$o}[0]) eq "mtf_ast") {
foreach $i (@{$o}) {
    $i->print() if (ref($i) eq "mtf_ast");
}
            } else {
printf "${curr_indent}$prop: \n";
foreach $i (@{$o}) {
    print "${curr_indent} $i\n";
}
    }
        } else {
            print "${curr_indent}$prop: $o \n";
        }
    }
    --$curr_level;
}
1;
```

A.1.3 mtf_client_parser.pm

```
package mtf_client_parser;
use mtf_ast;
use Carp;
use strict;
my $line;

#----------------------------------------------------------------------------
#
# Parse() creates an abstract syntax tree and returns the root object
# of the mtf_ast in $ROOT.
#
#----------------------------------------------------------------------------
sub parse{
    my ($package, $filename) = @_;
    open (P, $filename) || die "Could not open $filename : $@";
    my $c;
    my $d;
    my $rlist;
    my $label;
    my $name;
    my $cname;
    my $code_name;
    my $code_label;
```

```
    my $root = mtf_ast->new("Root");


    while (get_line()) {
# parse simple definitions
# if ($line =~ /^\s*set +(\w+) +(\w+)/) {
if ($line =~ /^\s*set +(\w+) +(.*)/) {
    $root->add_prop($1, $2);
}

# parse data blocks (single data definitions)
elsif ($line =~ /^\s*data +(\w+) +(\w+) +{/) {
    print "===tgen===> Parsing data block\n" if $main::verbose;
    $label = $1;
    $name  = $2;
    $c = mtf_ast->new($label);
    $c->add_prop("name", $name);
    $c->add_prop("type", "data");

    $rlist = [];
    while (get_line()) {
last if $line =~ /^\s*}/;
        $line =~ s/^\s*//;
        push(@$rlist,$line);
            }
            $c->add_prop("list", $rlist);
            #$root->add_prop_list("", $c);
            $root->add_prop($label, $c);
        }

# parse code blocks (single code definitions)
elsif ($line =~ /^\s*code +(\w+) +(\w+) +{/) {
    print "===tgen===> Parsing code block\n" if $main::verbose;
    $label = $1;
    $cname  = $2;
    $c = mtf_ast->new($label);
    $c->add_prop("cname", $cname);
    $c->add_prop("type", "code");

    while (get_line()) {
if ($line =~ /^\s*(\w+) +{/) {
    $code_label = $1;
    $d = mtf_ast->new($code_label);
    $d->add_prop("label", $code_label);

    $rlist = [];
    while (get_line()) {
last if $line =~ /^\s*}/;
        $line =~ s/^\s*//;
        push(@$rlist,$line);
    }
                $d->add_prop("list", $rlist);
    $c->add_prop($code_label, $d);
}
elsif ($line =~ /^\s*set +(\w+) +(.*)/) {
    $c->add_prop($1, $2);
}

last if ($line =~ /^s*}/);
            }

            $root->add_prop($label, $c);
```

```
        }

# parse codel blocks (multiple code definitions)
elsif ($line =~ /^\s*codel +(\w+) +(\w+) +{/) {
    print "===tgen===> Parsing codel block\n" if $main::verbose;
    $label = $1;
    $cname = $2;
    $c = mtf_ast->new($label);
    $c->add_prop("cname", $cname);
    $c->add_prop("type", "codel");

    while (get_line()) {
if ($line =~ /^\s*(\w+) +(\w+) +{/) {
    $code_label = $1;
    $code_name  = $2;
    $d = mtf_ast->new($code_label);
    $d->add_prop("label", $code_label);
    $d->add_prop("name", $code_name);

    $rlist = [];
    while (get_line()) {
last if $line =~ /^\s*}/;
        $line =~ s/^\s*//;
        push(@$rlist,$line);
    }
                    $d->add_prop("list", $rlist);
    $c->add_prop($code_label, $d);
}
elsif ($line =~ /^\s*set +(\w+) +(.*)/) {
    $c->add_prop($1, $2);
}
last if ($line =~ /^s*}/);
            }

            $root->add_prop_list("$label", $c);
        }


    }
    return $root;
}

sub get_line {
    while (defined($line = <P>)) {
print $line if $main::debugging;
        chomp $line;
        #$line =~ s#//.*$##;          # remove comments
        $line =~ s/#.*$//;          # remove comments
        last if $line !~ /^\s*$/;  # return if not white-space
    }
    $line;
}

# end mtf_client_parser (return true)
1;
```

A.1.4 mtf_template_parser.pm

```
package mtf_template_parser;
use strict;
```

```
#-------------------------------------------------------------------------
# This package parses a template file in the format explained below, and
# translates it into Perl code.
#
# The template file recognizes the following directives ...
#   (keywords are case insensitive)
#   @OPENFILE <filename> [options] - closes the previous output file,
#         the new file.
#         Options:
#            -append - open the file in append mode
#            -no_overwrite - do not overwrite the file if it already exists.
#                  This is useful if you want to generate the file only once.
#            -only_if_different - puts all the output into a temp file, does a
#                  diff with the given file, and overwrites it if the two
#                  files differ - useful in a make environment, where you
#                  don't want to unnecessarily touch the file if the contents
#                  are the same, to preserve timestamps
#
#   @PERL <perl code> - Inserts the perl code in the output file untranslated
#   @VISIT <node>  visit a parse node
#   @VEND          leave a parse node
#   @FOREACH <var> [perl condition code] - iterates thru the array @var, using
#                  the iterator variable $var_i. The iteration works
#                  wherever the condition is true.
#
#   @END - terminates the loop
#   @//  - comment line, not reproduced in the intermediate perl file
#   All other lines in the template are left essentially untranslated.
#-------------------------------------------------------------------------

sub parse {
    # Args : template file, intermediate perl file
    my ($pkg,$template_file, $inter_file) = @_;
    unless (open (T, $template_file)) {
        warn "$template_file : $@";
        return 1;
    }
    open (I, "> $inter_file") ||
        die "Error opening intermediate file $inter_file : $@";

    emit_opening_stmts($template_file);
    my $line;
    while (defined($line = <T>)) {
        if ($line !~ /^\s*\@/) { # Is it a command?
            emit_text($line);
            next;
        }
        if ($line =~ /^\s*\@OPENFILE\s*(.*)\s*$/i) {
            emit_open_file ($1);
        } elsif ($line =~ /^\s*\@FOREACH\s*(\w*)\s*(.*)\s*/i) {
            emit_loop_begin ($1,$2);
        } elsif ($line =~ /^\s*\@END/i) {
            emit_loop_end();
        } elsif ($line =~ /^\s*\@VISIT\s*(\w*)\s*(.*)\s*/i) {
            emit_visit_begin ($1,$2);
        } elsif ($line =~ /^\s*\@VEND/i) {
            emit_visit_end();
        } elsif ($line =~ /^\s*\@PERL(.*)/i) {
            emit_perl("$1\n");
        };
    }
    emit_closing_stmts();
```

```
    close(I);
    return 0;
}


# All pieces of output code are within a "here" document terminated
# by _EOC_
#

#----------------------------------------------------------------------
# emit_opening_stmts
# ==> emit ("Convert ROOT's properties to global variable names")
#
sub emit_opening_stmts {
    my $template_file = shift;
    emit("# Created automatically from $template_file");
    emit(<<'_EOC_');

use mtf_ast;
use mtf_util;

$tmp_file = "mtf.tmp";

sub open_file;
if (! (defined ($ROOT) && $ROOT)) {
    die "ROOT not defined";
}

$file = "> -";
open (F, $file) || die $@;
$code = "";
$ROOT->visit();
_EOC_
}

#------------------------------------------------------------------------
# emit_open_file
# ==> emit ("Close the previous file, and open the new filename for output
#

sub emit_open_file {
    my $file = shift;
    my $no_overwrite      = ($file =~ s/-no_overwrite//gi) ? 1 : 0;
    my $append            = ($file =~ s/-append//gi) ? 1 : 0;
    my $only_if_different = ($file =~ s/-only_if_different//gi) ? 1 : 0;
    $file =~ s/\s*//g;
    emit (<<"_EOC_");
# Line $.
open_file(\"$file\", $no_overwrite, $only_if_different, $append);
_EOC_
}


#----------------------------------------------------------------------
# emit_loop_begin
# ==> emit ("manufacture an iterator name, and visit each element in
#           that array")
# The best way to understand this code is to execute the schema compiler
# and look at the intermediate perl code.
#

sub emit_loop_begin {
    my $l_name = shift; # Name of the list variable
```

```
    my $condition = shift;
    my $l_name_i = $l_name . "_i";
emit (<<"_EOC_");
# Line $.
foreach \$$l_name_i (\@\${$l_name}) {
    \$$l_name_i->visit ();
_EOC_
    if ($condition) {
    emit ("next if (! ($condition));\n");
    }
}

#----------------------------------------------------------------------
sub emit_loop_end {
    emit(<<"_EOC_");
# Line $.
    mtf_ast->bye();
}
_EOC_
}

#----------------------------------------------------------------------
# emit_visit_begin
# ==> emit ("manufacture an iterator name, and visit each element in
#            that array")
# The best way to understand this code is to execute the schema compiler
# and look at the intermediate perl code.
#

sub emit_visit_begin {
    my $v_name = shift; # Name of the list variable
    my $condition = shift;
emit (<<"_EOC_");
# Line $.
    if (!defined(\$$v_name)) {
print "mtf: node '$v_name' not defined\\n";
exit 1;
    }
    \$$v_name->visit ();
_EOC_
    # GDB: check
    if ($condition) {
    emit ("next if (! ($condition));\n");
    }
}

#----------------------------------------------------------------------
sub emit_visit_end {
    emit(<<"_EOC_");
# Line $.
    mtf_ast->bye();
_EOC_
}

#----------------------------------------------------------------------
sub emit_perl {
    emit($_[0]);
}

#----------------------------------------------------------------------
sub emit_text {
    chomp $_[0];
    # Escape '\' in the text
```

```
    $_[0] =~ s/\\/\\\\/g;
    # Escape quotes in the text
    $_[0] =~ s/"/\\"/g;
    $_[0] =~ s/'/\\'/g;
    emit(<<"_EOC_");
output("$_[0]\\n");
_EOC_
}

#----------------------------------------------------------------------
sub emit_closing_stmts {
    emit(<<'_EOC_');
mtf_ast->bye();
close(F);
unlink ($tmp_file);

sub open_file {
    my ($a_file, $a_nooverwrite, $a_only_if_different, $a_append) = @_;

    #First deal with the file previously opened
    close (F);
    if ($only_if_different) {
        if (mtf_util::compare ($orig_file, $curr_file) != 0) {
            rename ($curr_file, $orig_file) ||
            die "Error renaming $curr_file  to $orig_file";
        }
    }

    #Now for the new file ...
    $curr_file = $orig_file = $a_file;
    $only_if_different = ($a_only_if_different && (-f $curr_file)) ? 1 : 0;
    $no_overwrite = ($a_nooverwrite && (-f $curr_file))  ? 1 : 0;
    $mode =  ($a_append) ? ">>" : ">";

    if ($only_if_different) {
        unlink ($tmp_file);
        $curr_file = $tmp_file;
    }

    if (! $no_overwrite) {
        open (F, "$mode $curr_file") || die "could not open $curr_file";
    }
}

sub output {
    print F @_ if (! $no_overwrite)
}
1;
_EOC_
}

#----------------------------------------------------------------------
sub emit {
    print I $_[0];
}

1; # returns 1 if successfully compiled
```

A.1.5 mtf_util.pm

```
package mtf_util;

sub compare {
    $f1 = $_[0]; $f2 = $_[1];
        if ((-s $f1) != (-s $f2)) {
        return 1;
    }
    open (F1, "$f1") || die "Could not open $f1";
    open (F2, "$f2") || die "Could not open $f2";
    while (sysread(F1, $buf1, 1024)) {
        return 1 if (! sysread(F2, $buf2, 1024)) ;
        return 1 if ($buf1 ne $buf2);
    }
    close (F1); close (F2);
    return 0;
}

1;
```

A.2 Target Templates

A.2.1 mtf_ds.tpl

```
@//
@// mtf_ds.tpl
@//
@// template for user-level direct switch (ds) threads
@//
@//----------------------------------------------------------------------
@//
@// define global variables
@//
@//----------------------------------------------------------------------
@visit sched
@perl $mtf_tcb_container = $cname;
@perl $mtf_tcb_container_init = $init;
@perl $mtf_tcb_put  = $put;
@perl $mtf_tcb_get  = $get;
@vend
@//
@perl $user = $ENV{"USER"};
@//
@//
@//----------------------------------------------------------------------
@//
@// Generate header file
@//
@//----------------------------------------------------------------------
@perl print "Generating ${name}.h\n";
@openfile ${name}.h
```

```
/*
 * ${name}.h
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */

#ifndef _${name}_h_
#define _${name}_h_

/* include files */
#include <assert.h>

/* macros */

@visit tcb
@perl $mtf_tcb = $name;
#define mtf_scb_ptr mtf_scb_current
#define mtf_scb_field(x) mtf_scb_ptr->x
#define mtf_scb_set(newscb) mtf_scb_ptr = newscb

#define mtf_ecb_ptr mtf_ecb_current
#define mtf_ecb_field(x) mtf_ecb_ptr->x
#define mtf_ecb_set(newecb) mtf_ecb_ptr = newecb

#define mtf_tcb_ptr mtf_ecb_field(current)
#define mtf_tcb_field(x) (mtf_tcb_ptr)->x
#define mtf_tcb_set(newtcb) mtf_tcb_ptr = newtcb

#define mtf_tcb_current() mtf_tcb_ptr
@vend


/* datatypes */

/* locks */
#define mtf_lock                int
#define mtf_lock_init(lock)
#define mtf_lock_free(lock)
#define mtf_lock_acquire(lock)
#define mtf_lock_release(lock)

/* thread control block (tcb) datatypes */

/* tcb function pointer */
typedef void * (mtf_tcb_func)(void *arg);

/* thread control block */
@visit tcb
@perl $mtf_tcb = $name;
typedef struct $name {
        /* client supplied tcb fields */
        @perl for (@$list) { output("\t$_\n"); }

        /* target fields */
        void *stack;
        void *sp;
} $name;
@vend

/* thread control block containers */
```

```
@visit tcb_container
typedef struct $name {
	@perl for (@$list) { output("\t$_\n"); }
} $name;
@vend

/* execution control block */
typedef struct mtf_ecb
{
	$mtf_tcb *current;
	$mtf_tcb *idle;
	$mtf_tcb *exit_tcb;
	$mtf_tcb *exiting;
} mtf_ecb;


typedef $mtf_tcb * (mtf_tcb_alloc_func)(void);
typedef void (mtf_tcb_free_func)($mtf_tcb *);
typedef void * (mtf_stack_alloc_func)(size_t);
typedef void (mtf_stack_free_func)(void *);

typedef struct mtf_scb
{
	mtf_tcb_alloc_func *tcb_alloc;
	mtf_tcb_free_func *tcb_free;
	mtf_stack_alloc_func *stack_alloc;
	mtf_stack_free_func *stack_free;
	$mtf_tcb_container *ready_container;
} mtf_scb;


/* global data */

extern unsigned mtf_stacksize;
extern mtf_scb *mtf_scb_current;
extern mtf_ecb *mtf_ecb_current;


/* functions */

/* initialization and system functions */
int  mtf_init(mtf_tcb_alloc_func *tcb_alloc,
              mtf_tcb_free_func *tcb_free,
              mtf_stack_alloc_func *stack_alloc,
              mtf_stack_free_func *stack_free);
int  mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg);
void mtf_exit(int status);
int  mtf_error(char *string, int code);

/* core TCB functions */
int mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg);
int mtf_tcb_start($mtf_tcb *tcb);
int mtf_tcb_yield(void);
void mtf_tcb_exit(void);

/* malloc and free */
#define mtf_malloc(size) malloc(size)
#define mtf_free(ptr) free(ptr)

/* include client header file */
#include"${client_inc}"
```

```
#endif /* _${name}_h_ */
@//
@//
@//------------------------------------------------------------------------
@//
@// Generate C file
@//
@//------------------------------------------------------------------------
@perl print "Generating ${name}.c\n";
@openfile ${name}.c
/*
 * ${name}.c
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */

#include<stdio.h>
#include<stdlib.h>
#include"${name}.h"


/* constants */

unsigned mtf_stacksize = 32768;


/* data */

mtf_scb *mtf_scb_current;
mtf_ecb *mtf_ecb_current;


/* interface functions */

void mtf_ds_exit();

#define GRAN 8                  /* alignment granularity */
#define PS_ALIGN(x)  (((x)+GRAN-1) & ~(GRAN-1))

/* error reporting and exiting function */

int
mtf_error(char *s, int code)
{
        printf("mtf error: %s\n", s);
        exit(code);
}


/* use srt_stk_underflow to exit thread */

void
ds_stk_underflow(void)
{
        mtf_tcb_exit();
}


void
ds_stk_overflow(void)
```

```
{
        mtf_error("ds_stk_overflow called -- exiting", 1);
}


void
ds_stk_corrupted(void)
{
        mtf_error("ds_stk_corrupted called -- exiting", 1);
}



/* template functions */


int
mtf_init(mtf_tcb_alloc_func *tcb_alloc, mtf_tcb_free_func *tcb_free,
         mtf_stack_alloc_func *stack_alloc, mtf_stack_free_func *stack_free)
{
        /* allocate a system control block */
        mtf_scb_set((mtf_scb *) malloc(sizeof(mtf_scb)));
        mtf_scb_field(tcb_alloc)   = tcb_alloc;
        mtf_scb_field(tcb_free)    = tcb_free;
        mtf_scb_field(stack_alloc) = stack_alloc;
        mtf_scb_field(stack_free)  = stack_free;
        mtf_scb_field(ready_container) =
            ($mtf_tcb_container *) malloc(sizeof($mtf_tcb_container));

        /* initialize the ready container */
        $mtf_tcb_container_init(mtf_scb_field(ready_container));

        /* allocate an execution control block */
        mtf_ecb_set((mtf_ecb *) malloc(sizeof(mtf_ecb)));

        /* initialize exit tcb */
        mtf_ecb_field(exit_tcb) = mtf_scb_field(tcb_alloc)();
        mtf_ecb_field(exit_tcb)->stacksize = mtf_stacksize;
        mtf_tcb_init(mtf_ecb_field(exit_tcb), (mtf_tcb_func *) mtf_ds_exit, (void *) 0);

        return 0;
}


int
mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg)
{
        /* initialize main tcb */
        main_tcb->next = NULL;
        main_tcb->stack = NULL;
        main_tcb->sp = NULL;

        mtf_tcb_init(main_tcb, main_func , main_arg);

        mtf_tcb_set(main_tcb);

        ds_chg_context(main_tcb->sp, NULL);

        /* not reached */
        return 1;
}
```

```
int
mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg)
{
        tcb->stack = (mtf_scb_field(stack_alloc))(tcb->stacksize);
        tcb->sp = (void *) PS_ALIGN((long) tcb->stack);
        ds_build_context(func, tcb->sp, tcb->stacksize, arg, 0L, 0L, 0L);
}


int
mtf_tcb_start($mtf_tcb *tcb)
{
        $mtf_tcb_put(tcb, mtf_scb_field(ready_container));

        return 1;
}


int
mtf_tcb_yield(void)
{
        $mtf_tcb *old, *new;

        /* get next tcb to schedule */
        if ((new = $mtf_tcb_get(mtf_scb_field(ready_container))) != NULL) {
                old = mtf_ecb_field(current);
                $mtf_tcb_put(old, mtf_scb_field(ready_container));
                mtf_tcb_set(new);
                ds_chg_context(new->sp, old->sp);
        }
        /* No tcbs to schedule - return to calling tcb */
}


void mtf_ds_exit(void)
{
        $mtf_tcb *old, *new;

        while (1) {
                /* Deallocate tcb */
                if (mtf_ecb_field(exiting)->stack) {
                        mtf_scb_field(stack_free)(mtf_ecb_field(exiting)->stack);
                }
                if (mtf_ecb_field(exiting)) {
                        mtf_scb_field(tcb_free)(mtf_ecb_field(exiting));
                }

                mtf_ecb_field(exiting) = NULL;

                /* Get next tcb to schedule */
                if ((new = $mtf_tcb_get(mtf_scb_field(ready_container))) != NULL) {
                        old = mtf_ecb_field(current);
                        mtf_tcb_set(new);
                        ds_chg_context(new->sp, old->sp);
                } else {
                        /* No tcbs to schedule */
                        /* mtf_error("mtf_ds_exit: no more tcbs", 1); */
                        mtf_exit(1);
                }
        }
}
```

```
void
mtf_tcb_exit(void)
{
        mtf_ecb_field(exiting) = mtf_tcb_ptr;
        mtf_tcb_set(mtf_ecb_field(exit_tcb));
        ds_chg_context(mtf_ecb_field(exit_tcb)->sp, NULL);
        /* never return */
}


void
mtf_exit(int status)
{
        fflush(stdout);
        exit(status);
}


@foreach sync
@visit block
/*
 * block current tcb and schedule a new tcb
 */
int
$name ($cname *b_container, mtf_lock *lock)
{
        $mtf_tcb *b_tcb, *s_tcb;

        /* get next tcb to schedule */
        s_tcb = $mtf_tcb_get(mtf_scb_field(ready_container));
        b_tcb = mtf_tcb_ptr;

        if (s_tcb == NULL) {
                /* no tcbs to schedule */
                /* mtf_error("mtf_ds_block: no more tcbs", 1); */
                mtf_exit(1);
        }

        @perl for (@$list) { output("\t$_\n"); }

        mtf_tcb_set(s_tcb);

        /* switch to new tcb */
        ds_chg_context(s_tcb->sp, b_tcb->sp);
}
@vend

@visit unblock
/*
 * unblock a tcb
 */
int
$name ($cname *b_container)
{
        $mtf_tcb *b_tcb;
        /* get tcb from container */
        @perl for (@$list) { output("\t$_\n"); }

        /* put tcb on ready container */
        if (b_tcb != NULL) {
               $mtf_tcb_put(b_tcb, mtf_scb_field(ready_container));
        }
```

```
}
@vend
@end
```

A.2.2 mtf_ps.tpl

```
@//
@// mtf_ps.tpl
@//
@// template for user-level preswitch (ps) threads (based on QuickThreads)
@//
@//------------------------------------------------------------------------
@//
@// define global variables
@//
@//------------------------------------------------------------------------
@visit sched
@perl $mtf_tcb_container = $cname;
@perl $mtf_tcb_container_init = $init;
@perl $mtf_tcb_put  = $put;
@perl $mtf_tcb_get  = $get;
@vend
@//
@perl $user = $ENV{"USER"};
@//
@//
@//------------------------------------------------------------------------
@//
@// Generate header file
@//
@//------------------------------------------------------------------------
@perl print "Generating ${name}.h\n";
@openfile ${name}.h
/*
 * ${name}.h
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */

#ifndef _${name}_h_
#define _${name}_h_

/* include files */
#include <assert.h>
#include <mtf/qt.h>

/* macros */

@visit tcb
@perl $mtf_tcb = $name;
#define mtf_scb_ptr mtf_scb_current
#define mtf_scb_field(x) mtf_scb_ptr->x
#define mtf_scb_set(newscb) mtf_scb_ptr = newscb

#define mtf_ecb_ptr mtf_ecb_current
#define mtf_ecb_field(x) mtf_ecb_ptr->x
#define mtf_ecb_set(newecb) mtf_ecb_ptr = newecb
```

```
#define mtf_tcb_ptr mtf_ecb_field(current)
#define mtf_tcb_field(x) (mtf_tcb_ptr)->x
#define mtf_tcb_set(newtcb) mtf_tcb_ptr = newtcb

#define mtf_tcb_current() mtf_tcb_ptr
@vend


#define MTF_QT_STKALIGN(sp, alignment) \
  ((void *)((((qt_word_t)(sp)) + (alignment) - 1) & ~((alignment)-1)))


/* datatypes */

/* locks */
#define mtf_lock                int
#define mtf_lock_init(lock)
#define mtf_lock_free(lock)
#define mtf_lock_acquire(lock)
#define mtf_lock_release(lock)

/* thread control block (tcb) datatypes */

/* tcb function pointer */
typedef void * (mtf_tcb_func)(void *arg);

/* thread control block */
@visit tcb
@perl $mtf_tcb = $name;
typedef struct $name {
        /* client supplied tcb fields */
        @perl for (@$list) { output("\t$_\n"); }

        /* target fields */
        void *stack;
        void *aligned_stack;
        qt_t *sp;
} $name;
@vend

/* thread control block containers */
@visit tcb_container
typedef struct $name {
        @perl for (@$list) { output("\t$_\n"); }
} $name;
@vend

/* execution control block */
typedef struct mtf_ecb
{
        $mtf_tcb *current;
        $mtf_tcb *idle;
        $mtf_tcb *exitfunc;
        $mtf_tcb *exiting;
} mtf_ecb;


typedef $mtf_tcb * (mtf_tcb_alloc_func)(void);
typedef void (mtf_tcb_free_func)($mtf_tcb *);
typedef void * (mtf_stack_alloc_func)(size_t);
typedef void (mtf_stack_free_func)(void *);

typedef struct mtf_scb
```

```
{
        mtf_tcb_alloc_func *tcb_alloc;
        mtf_tcb_free_func *tcb_free;
        mtf_stack_alloc_func *stack_alloc;
        mtf_stack_free_func *stack_free;
        $mtf_tcb_container *ready_container;
} mtf_scb;


/* global data */

extern unsigned mtf_stacksize;
extern mtf_scb *mtf_scb_current;
extern mtf_ecb *mtf_ecb_current;


/* functions */

/* initialization and system functions */
int  mtf_init(mtf_tcb_alloc_func *tcb_alloc,
              mtf_tcb_free_func *tcb_free,
              mtf_stack_alloc_func *stack_alloc,
              mtf_stack_free_func *stack_free);
int  mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg);
void mtf_exit(int status);
int  mtf_error(char *string, int code);

/* core TCB functions */
int mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg);
int mtf_tcb_start($mtf_tcb *tcb);
int mtf_tcb_yield(void);
void mtf_tcb_exit(void);

/* malloc and free */
#define mtf_malloc(size) malloc(size)
#define mtf_free(ptr) free(ptr)

/* include client header file */
#include"${client_inc}"


#endif /* _${name}_h_ */
@//
@//
@//-----------------------------------------------------------------------------
@//
@// Generate C file
@//
@//-----------------------------------------------------------------------------
@perl print "Generating ${name}.c\n";
@openfile ${name}.c
/*
 * ${name}.c
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */

#include<stdio.h>
#include<stdlib.h>
#include"${name}.h"
```

```
/* constants */

unsigned mtf_stacksize = 32768;


/* data */

mtf_scb *mtf_scb_current;
mtf_ecb *mtf_ecb_current;


/* interface functions */

void mtf_qt_exit(void);


/*-------------------------------------------------------------------------*
 * Common functions                                                        *
 *-------------------------------------------------------------------------*/

/* error reporting and exiting function */

int
mtf_error(char *s, int code)
{
        printf("mtf error: %s\n", s);
        exit(code);
}

/*-------------------------------------------------------------------------*
 * Template functions                                                      *
 *-------------------------------------------------------------------------*/


int
mtf_init(mtf_tcb_alloc_func *tcb_alloc, mtf_tcb_free_func *tcb_free,
         mtf_stack_alloc_func *stack_alloc, mtf_stack_free_func *stack_free)
{
        /* allocate a system control block */
        mtf_scb_set((mtf_scb *) malloc(sizeof(mtf_scb)));
        mtf_scb_field(tcb_alloc)   = tcb_alloc;
        mtf_scb_field(tcb_free)    = tcb_free;
        mtf_scb_field(stack_alloc) = stack_alloc;
        mtf_scb_field(stack_free)  = stack_free;
        mtf_scb_field(ready_container) =
            ($mtf_tcb_container *) malloc(sizeof($mtf_tcb_container));

        /* initialize the ready container */
        $mtf_tcb_container_init(mtf_scb_field(ready_container));

        /* allocate an execution control block */
        mtf_ecb_set((mtf_ecb *) malloc(sizeof(mtf_ecb)));

        return 0;
}


void
mtf_init_helper(qt_t *sp, void *old, void *blockq)
{
        /* do nothing */
```

```
}


int
mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg)
{
        /* initialize main tcb */
        main_tcb->next = NULL;
        main_tcb->stack = NULL;
        main_tcb->sp = NULL;

        mtf_tcb_init(main_tcb, main_func , main_arg);

        mtf_tcb_set(main_tcb);

        QT_BLOCK((qt_helper_t *) mtf_init_helper, NULL, NULL, main_tcb->sp);

        /* not reached */
        return 1;
}


void
mtf_create_prefix(void *u, void *t, qt_userf_t *f)
{
        mtf_tcb_set(($mtf_tcb *) t);
        ((mtf_tcb_func *)f)(u);

        /* exit with no error */
        mtf_tcb_exit();

        /* NOT REACHED */
}


int
mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg)
{
        /* allocate stack */
        tcb->stack = (mtf_scb_field(stack_alloc))(tcb->stacksize);

        /* align the stack */
        tcb->aligned_stack = MTF_QT_STKALIGN(tcb->stack, QT_STKALIGN);

        /* Get the beginning of the stack */
        tcb->sp = QT_SP (tcb->aligned_stack, mtf_stacksize - QT_STKALIGN);

        /* initialize the stack */
        tcb->sp = QT_ARGS (tcb->sp, arg, tcb, (qt_userf_t *) func,
                           mtf_create_prefix);

        return 0;
}


int
mtf_tcb_start($mtf_tcb *tcb)
{
        $mtf_tcb_put(tcb, mtf_scb_field(ready_container));

        return 1;
}
```

```
void
mtf_yield_helper(qt_t *sp, void *old, void *blockq)
{
        (($mtf_tcb *) old)->sp = sp;
        $mtf_tcb_put(($mtf_tcb *) old, ($mtf_tcb_container *) blockq);
}


int
mtf_tcb_yield(void)
{
        $mtf_tcb *old, *new;

        /* get next tcb to schedule */
        if ((new = $mtf_tcb_get(mtf_scb_field(ready_container))) != NULL) {
                 /* the ready queue is not empty */
                 old = mtf_ecb_field(current);
                 mtf_tcb_set(new);
                 QT_BLOCK((qt_helper_t *) mtf_yield_helper, old,
                        (void *) mtf_scb_field(ready_container), new->sp);
        }
        /* No tcbs to schedule - return to calling tcb */
}


void
mtf_exit_helper(qt_t *sp, void *exiting, void *blockq)
{
        /* Deallocate tcb */

        assert((($mtf_tcb *) exiting) != NULL);

        if ( (($mtf_tcb *) exiting)->stack ) {
                mtf_scb_field(stack_free)( (($mtf_tcb *) exiting)->stack );
        }
        if ( ($mtf_tcb *) exiting) {
                mtf_scb_field(tcb_free)( ($mtf_tcb *) exiting);
        }
}


void mtf_tcb_exit(void)
{
        $mtf_tcb *old, *new;

        /* Get next tcb to schedule */
        if ((new = $mtf_tcb_get(mtf_scb_field(ready_container))) != NULL) {
                 old = mtf_ecb_field(current);
                 mtf_tcb_set(new);
                 QT_BLOCK((qt_helper_t *) mtf_exit_helper, old,
                         (void *) NULL, new->sp);
        } else {
                /* No TCBs to schedule */
                /* mtf_error("mtf_qt_exit: no more tcbs", 1); */
                mtf_exit(1);
        }
        /* never return */
}


void
mtf_exit(int status)
```

```
{
        fflush(stdout);
        exit(status);
}


@foreach sync
@visit block
/*
 * block helper
 */
int
${name}_helper(qt_t *sp, void *b_tcb, $cname *b_container)
{
        (($mtf_tcb *) b_tcb)->sp = sp;

        @perl for (@$list) { output("\t$_\n"); }
}


/*
 * block current tcb and schedule a new tcb
 */
int
$name ($cname *b_container, mtf_lock *lock)
{
        $mtf_tcb *b_tcb, *s_tcb;

        /* get next tcb to schedule */
        s_tcb = $mtf_tcb_get(mtf_scb_field(ready_container));
        b_tcb = mtf_tcb_ptr;

        if (s_tcb == NULL) {
                /* mtf_error("mtf_qt_exit: no more tcbs", 1); */
                mtf_exit(1);
        }

        mtf_tcb_set(s_tcb);

        /* switch to new tcb */
        QT_BLOCK((qt_helper_t *) ${name}_helper, b_tcb,
                 (void *) b_container, s_tcb->sp);
}
@vend

@visit unblock
/*
 * unblock a tcb
 */
int
$name ($cname *b_container)
{
        $mtf_tcb *b_tcb;
        /* get tcb from container */
        @perl for (@$list) { output("\t$_\n"); }

        assert(b_tcb != NULL);

        /* put tcb on ready container */
        if (b_tcb != NULL) {
               $mtf_tcb_put(b_tcb, mtf_scb_field(ready_container));
        }
```

```
}
@vend
@end
```

A.2.3 mtf_pspr.tpl

```
@//
@// mtf_pspr.tpl
@//
@// template for user-level preswitch preemptive (pspr) threads
@//
@//-----------------------------------------------------------------------
@//
@// define global variables
@//
@//-----------------------------------------------------------------------
@visit sched
@perl $mtf_tcb_container = $cname;
@perl $mtf_tcb_container_init = $init;
@perl $mtf_tcb_put  = $put;
@perl $mtf_tcb_get  = $get;
@vend
@//
@perl $user = $ENV{"USER"};
@//
@//
@//-----------------------------------------------------------------------
@//
@// Generate header file
@//
@//-----------------------------------------------------------------------
@perl print "Generating ${name}.h\n";
@openfile ${name}.h
/*
 * ${name}.h
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */

#ifndef _${name}_h_
#define _${name}_h_

/* include files */
#include <assert.h>
#include <mtf/qt.h>

/* macros */

@visit tcb
@perl $mtf_tcb = $name;
#define mtf_scb_ptr mtf_scb_current
#define mtf_scb_field(x) mtf_scb_ptr->x
#define mtf_scb_set(newscb) mtf_scb_ptr = newscb

#define mtf_ecb_ptr mtf_ecb_current
#define mtf_ecb_field(x) mtf_ecb_ptr->x
#define mtf_ecb_set(newecb) mtf_ecb_ptr = newecb

#define mtf_tcb_ptr mtf_ecb_field(current)
```

```
#define mtf_tcb_field(x) (mtf_tcb_ptr)->x
#define mtf_tcb_set(newtcb) mtf_tcb_ptr = newtcb

#define mtf_tcb_current() mtf_tcb_ptr
@vend


#define MTF_QT_STKALIGN(sp, alignment) \
  ((void *)((((qt_word_t)(sp)) + (alignment) - 1) & ~((alignment)-1)))


/* datatypes */

/* locks */
#define mtf_lock                int
#define mtf_lock_init(lock)
#define mtf_lock_free(lock)
#define mtf_lock_acquire(lock)  (mtf_ecb_field(preempt_mask))++
#define mtf_lock_release(lock)  { \
        (mtf_ecb_field(preempt_mask))--;                   \
        if (mtf_ecb_field(preempt_mask) == 0) {            \
               if (mtf_ecb_field(preempt_state == 1)) {    \
                       mtf_ecb_field(preempt_state) = 0;   \
                       mtf_tcb_yield();                    \
               }                                           \
        }                                                  \
  }


/* thread control block (tcb) datatypes */

/* tcb function pointer */
typedef void * (mtf_tcb_func)(void *arg);

/* thread control block */
@visit tcb
@perl $mtf_tcb = $name;
typedef struct $name {
        /* client supplied tcb fields */
        @perl for (@$list) { output("\t$_\n"); }

        /* target fields */
        void *stack;
        void *aligned_stack;
        qt_t *sp;
} $name;
@vend

/* thread control block containers */
@visit tcb_container
typedef struct $name {
        @perl for (@$list) { output("\t$_\n"); }
} $name;
@vend

/* execution control block */
typedef struct mtf_ecb
{
        $mtf_tcb *current;
        $mtf_tcb *idle;
        $mtf_tcb *exitfunc;
        $mtf_tcb *exiting;
        int preempt_mask;
```

```
        int preempt_state;
} mtf_ecb;


typedef $mtf_tcb * (mtf_tcb_alloc_func)(void);
typedef void (mtf_tcb_free_func)($mtf_tcb *);
typedef void * (mtf_stack_alloc_func)(size_t);
typedef void (mtf_stack_free_func)(void *);

typedef struct mtf_scb
{
        mtf_tcb_alloc_func *tcb_alloc;
        mtf_tcb_free_func *tcb_free;
        mtf_stack_alloc_func *stack_alloc;
        mtf_stack_free_func *stack_free;
        $mtf_tcb_container *ready_container;
} mtf_scb;


/* global data */

extern unsigned mtf_stacksize;
extern mtf_scb *mtf_scb_current;
extern mtf_ecb *mtf_ecb_current;


/* functions */

/* initialization and system functions */
int  mtf_init(mtf_tcb_alloc_func *tcb_alloc,
              mtf_tcb_free_func *tcb_free,
              mtf_stack_alloc_func *stack_alloc,
              mtf_stack_free_func *stack_free);
int  mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg);
void mtf_exit(int status);
int  mtf_error(char *string, int code);

/* core TCB functions */
int mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg);
int mtf_tcb_start($mtf_tcb *tcb);
int mtf_tcb_yield(void);
void mtf_tcb_exit(void);

/* malloc and free */
#define mtf_malloc(size) mtf_pr_malloc(size)
#define mtf_free(ptr) mtf_pr_free(ptr)

/* include client header file */
#include"${client_inc}"


#endif /* _${name}_h_ */
@//
@//
@//------------------------------------------------------------------------
@//
@// Generate C file
@//
@//------------------------------------------------------------------------
@perl print "Generating ${name}.c\n";
@openfile ${name}.c
/*
 * ${name}.c
```

```
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */

#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<sys/time.h>
#include"${name}.h"


/* constants */

unsigned mtf_stacksize = 32768;


/* data */

mtf_scb *mtf_scb_current;
mtf_ecb *mtf_ecb_current;


/* global debug data */
#ifdef MTF_PROFILE
static long mtf_pspr_prcount  = 0;
static long mtf_pspr_ctxcount = 0;
#endif


/* interface functions */

void mtf_qt_exit(void);


/*------------------------------------------------------------------------*
 * Common functions                                                       *
 *------------------------------------------------------------------------*/

/* error reporting and exiting function */

int
mtf_error(char *s, int code)
{
        printf("mtf error: %s\n", s);
        exit(code);
}

/*------------------------------------------------------------------------*
 * Template functions                                                     *
 *------------------------------------------------------------------------*/


void *
mtf_pr_malloc(size_t size)
{
        /* preemption-safe malloc */

        void *rv;

        mtf_lock_acquire(INTS);
```

```
        rv = (void *) malloc(size);
        mtf_lock_release(INTS);

        return rv;
}


void
mtf_pr_free(void *ptr)
{
        /* preemption-safe free */

        mtf_lock_acquire(INTS);
        free(ptr);
        mtf_lock_release(INTS);
}


int
mtf_init_preemption(void)
{
        struct sigaction snew, sold;
        struct itimerval quantum, quantum_old;

        /* initialize preemption fields */
        mtf_ecb_field(preempt_mask) = 0;
        mtf_ecb_field(preempt_state) = 0;
}


void
mtf_preemption_handler(int signo)
{
#ifdef MTF_PROFILE
        mtf_pspr_prcount++;
#endif
        if (mtf_ecb_field(preempt_mask) > 0) {
                mtf_ecb_field(preempt_state) = 1;
        } else {
#ifdef MTF_PROFILE
                mtf_pspr_ctxcount++;
#endif
                mtf_tcb_yield();
        }

        return;
}


int
mtf_start_preemption(void)
{
        struct sigaction snew, sold;
        struct itimerval it_quantum, it_quantum_old;
        char *s;
        long quantum = 10000;


        /* get quantum from environment variable if it exists */
        if (s = getenv("MTF_PR_QUANTUM")) {
                quantum = atol(s);

        }
```

```
        if (quantum > 100000000) {
                printf("mtf_start_preemption: quantum (%d) too large", quantum);
                exit(1);
        }

        /* setup timer */
        it_quantum.it_interval.tv_sec = 0;
        it_quantum.it_interval.tv_usec = quantum;

        it_quantum.it_value.tv_sec = 0;
        it_quantum.it_value.tv_usec = quantum;

        setitimer(ITIMER_VIRTUAL, &it_quantum, &it_quantum_old);

        /* setup signal handler */

        snew.sa_handler = mtf_preemption_handler;
        sigemptyset(&snew.sa_mask);
        snew.sa_flags = SA_NODEFER;

        sigaction(SIGVTALRM, &snew, &sold);
}


int
mtf_init(mtf_tcb_alloc_func *tcb_alloc, mtf_tcb_free_func *tcb_free,
         mtf_stack_alloc_func *stack_alloc, mtf_stack_free_func *stack_free)
{
        /* allocate a system control block */
        mtf_scb_set((mtf_scb *) malloc(sizeof(mtf_scb)));
        mtf_scb_field(tcb_alloc)   = tcb_alloc;
        mtf_scb_field(tcb_free)    = tcb_free;
        mtf_scb_field(stack_alloc) = stack_alloc;
        mtf_scb_field(stack_free)  = stack_free;
        mtf_scb_field(ready_container) =
            ($mtf_tcb_container *) malloc(sizeof($mtf_tcb_container));

        /* initialize the ready container */
        $mtf_tcb_container_init(mtf_scb_field(ready_container));

        /* allocate an execution control block */
        mtf_ecb_set((mtf_ecb *) malloc(sizeof(mtf_ecb)));

        /* start preemption */
        mtf_start_preemption();
}


void
mtf_init_helper(qt_t *sp, void *old, void *blockq)
{
        /* do nothing */
}


int
mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg)
{
        /* initialize main tcb */
        main_tcb->next = NULL;
        main_tcb->stack = NULL;
        main_tcb->sp = NULL;
```

```
        mtf_tcb_init(main_tcb, main_func , main_arg);

        mtf_tcb_set(main_tcb);

        /* initialize preemption */
        mtf_init_preemption();

        QT_BLOCK((qt_helper_t *) mtf_init_helper, NULL, NULL, main_tcb->sp);

        /* not reached */
        return 1;
}


void
mtf_create_prefix(void *u, void *t, qt_userf_t *f)
{
        mtf_tcb_set(($mtf_tcb *) t);
        ((mtf_tcb_func *)f)(u);

        /* exit with no error */
        mtf_tcb_exit();

        /* NOT REACHED */
}


int
mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg)
{
        /* allocate stack */
        tcb->stack = (mtf_scb_field(stack_alloc))(tcb->stacksize);

        /* align the stack */
        tcb->aligned_stack = MTF_QT_STKALIGN(tcb->stack, QT_STKALIGN);

        /* Get the beginning of the stack */
        tcb->sp = QT_SP (tcb->aligned_stack, mtf_stacksize - QT_STKALIGN);

        /* initialize the stack */
        tcb->sp = QT_ARGS (tcb->sp, arg, tcb, (qt_userf_t *) func,
                           mtf_create_prefix);

        return 0;
}


int
mtf_tcb_start($mtf_tcb *tcb)
{
        mtf_lock_acquire(INTS);
        $mtf_tcb_put(tcb, mtf_scb_field(ready_container));
        mtf_lock_release(INTS);
        return 1;
}


void
mtf_yield_helper(qt_t *sp, void *old, void *blockq)
{
        (($mtf_tcb *) old)->sp = sp;
        $mtf_tcb_put(($mtf_tcb *) old, ($mtf_tcb_container *) blockq);
```

```
        mtf_lock_release(INTS);
}


int
mtf_tcb_yield(void)
{
        $mtf_tcb *old, *new;

        /* get next tcb to schedule */

        mtf_lock_acquire(INTS);
        new = $mtf_tcb_get(mtf_scb_field(ready_container));

        /* get next tcb to schedule */
        if (new != NULL) {
                /* the ready queue is not empty */
                old = mtf_ecb_field(current);
                mtf_tcb_set(new);
                QT_BLOCK((qt_helper_t *) mtf_yield_helper, old,
                        (void *) mtf_scb_field(ready_container), new->sp);
        } else {
                 /* No tcbs to schedule - return to calling tcb */
                 mtf_lock_release(INTS);
        }
}


void
mtf_exit_helper(qt_t *sp, void *exiting, void *blockq)
{
        /* Deallocate tcb */

        assert((($mtf_tcb *) exiting) != NULL);

        if ( (($mtf_tcb *) exiting)->stack ) {
                mtf_scb_field(stack_free)( (($mtf_tcb *) exiting)->stack );
        }

        mtf_lock_release(INTS);

        if ( ($mtf_tcb *) exiting) {
                mtf_scb_field(tcb_free)( ($mtf_tcb *) exiting);
        }
}


void mtf_tcb_exit(void)
{
        $mtf_tcb *old, *new;

        /* get next tcb to schedule */

        mtf_lock_acquire(INTS);
        new = $mtf_tcb_get(mtf_scb_field(ready_container));
        mtf_lock_release(INTS);


        if (new != NULL) {
                 old = mtf_ecb_field(current);
                 mtf_tcb_set(new);
                 QT_BLOCK((qt_helper_t *) mtf_exit_helper, old,
                         (void *) NULL, new->sp);
```

```
        } else {
                /* no TCBs to schedule */
                /* mtf_error("mtf_tcb_exit: no more tcbs", 1); */
                mtf_exit(1);
        }
        /* never return */
}


void
mtf_exit(int status)
{
#ifdef MTF_PROFILE
        printf("\n");
        printf("mtf_pspr: preemption count = %ld\n", mtf_pspr_prcount);
        printf("mtf_pspr: pr ctx count     = %ld\n", mtf_pspr_ctxcount);
#endif
        fflush(stdout);
        exit(status);
}


@foreach sync
@visit block
/*
 * block helper
 */
int
${name}_helper(qt_t *sp, void *b_tcb, $cname *b_container)
{
        (($mtf_tcb *) b_tcb)->sp = sp;

        @perl for (@$list) { output("\t$_\n"); }
        mtf_lock_release(INTS);
}


/*
 * block current tcb and schedule a new tcb
 */
int
$name ($cname *b_container, mtf_lock *lock)
{
        $mtf_tcb *b_tcb, *s_tcb;

        /* get next tcb to schedule */
        mtf_lock_acquire(INTS);
        s_tcb = $mtf_tcb_get(mtf_scb_field(ready_container));
        mtf_lock_release(INTS);

        b_tcb = mtf_tcb_ptr;

        if (s_tcb == NULL) {
                /* mtf_error("mtf_block: no more tcbs", 1); */
                mtf_exit(1);
        }

        mtf_tcb_set(s_tcb);

        /* switch to new tcb */
        QT_BLOCK((qt_helper_t *) ${name}_helper, b_tcb,
                 (void *) b_container, s_tcb->sp);
```

```
        /* re-acquire the container lock before returning to the client */
        /* pontentially this could be optimized */
        mtf_lock_acquire(INTS);
}
@vend

@visit unblock
/*
 * unblock a tcb
 */
int
$name ($cname *b_container)
{
        $mtf_tcb *b_tcb;
        /* get tcb from container */
        @perl for (@$list) { output("\t$_\n"); }

        assert(b_tcb != NULL);

        /* put tcb on ready container */
        if (b_tcb != NULL) {
                mtf_lock_acquire(INTS);
                $mtf_tcb_put(b_tcb, mtf_scb_field(ready_container));
                mtf_lock_release(INTS);
        }

}
@vend
@end
```

A.2.4 mtf_psmp.tpl

```
@//
@// mtf_psmp.tpl
@//
@// template for user-level preswitch multiprocessor (psmp) threads
@//
@//-----------------------------------------------------------------------
@//
@// define global variables
@//
@//-----------------------------------------------------------------------
@visit sched
@perl $mtf_tcb_container = $cname;
@perl $mtf_tcb_container_init = $init;
@perl $mtf_tcb_put  = $put;
@perl $mtf_tcb_get  = $get;
@perl $mtf_tcb_head = $head;
@vend
@//
@perl $user = $ENV{"USER"};
@//
@//
@//-----------------------------------------------------------------------
@//
@// Generate header file
@//
@//-----------------------------------------------------------------------
@perl print "Generating ${name}.h\n";
@openfile ${name}.h
/*
 * ${name}.h
```

```
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1999 by Gregory D. Benson
 */

#ifndef _${name}_h_
#define _${name}_h_

/* include files */
#include <assert.h>
#include <mtf/qt.h>
#include <mtf/system/mp.h>

/* macros */

@visit tcb
@perl $mtf_tcb = $name;
#define mtf_scb_ptr mtf_scb_current
#define mtf_scb_field(x) mtf_scb_ptr->x
#define mtf_scb_set(newscb) mtf_scb_ptr = newscb

#define mtf_ecb_ptr mtf_ecb_current[MTF_MP_GET_VPID]
#define mtf_ecb_field(x) mtf_ecb_ptr->x
#define mtf_ecb_set(newecb) mtf_ecb_ptr = newecb

#define mtf_tcb_ptr mtf_ecb_field(current)
#define mtf_tcb_field(x) (mtf_tcb_ptr)->x
#define mtf_tcb_set(newtcb) mtf_tcb_ptr = newtcb

#define mtf_tcb_current() mtf_tcb_ptr
@vend


#define MTF_QT_STKALIGN(sp, alignment) \
  ((void *)((((qt_word_t)(sp)) + (alignment) - 1) & ~((alignment)-1)))


/* datatypes */

/* locks */
#define mtf_lock                mtf_mp_lock_t
#define mtf_lock_init(lock)     mtf_mp_lock_init(lock)
#define mtf_lock_free(lock)     mtf_mp_lock_free(lock)
#define mtf_lock_acquire(lock)  mtf_mp_lock_acquire(lock)
#define mtf_lock_release(lock)  mtf_mp_lock_release(lock)

/* thread control block (tcb) datatypes */

/* tcb function pointer */
typedef void * (mtf_tcb_func)(void *arg);

/* thread control block */
@visit tcb
@perl $mtf_tcb = $name;
typedef struct $name {
        /* client supplied tcb fields */
        @perl for (@$list) { output("\t$_\n"); }

        /* target fields */
        void *stack;
        void *aligned_stack;
```

```
        qt_t *sp;
        mtf_lock *prev_clock;
} $name;
@vend

/* thread control block containers */
@visit tcb_container
typedef struct $name {
        @perl for (@$list) { output("\t$_\n"); }
} $name;
@vend

/* execution control block */
typedef struct mtf_ecb
{
        int      id;
        $mtf_tcb *current;
        $mtf_tcb *idle;
        /* $mtf_tcb *exitfunc; */
        /* $mtf_tcb *exiting; */
} mtf_ecb;


/* system control block */
typedef $mtf_tcb * (mtf_tcb_alloc_func)(void);
typedef void (mtf_tcb_free_func)($mtf_tcb *);
typedef void * (mtf_stack_alloc_func)(size_t);
typedef void (mtf_stack_free_func)(void *);

typedef struct mtf_scb
{
        mtf_tcb_alloc_func *tcb_alloc;
        mtf_tcb_free_func *tcb_free;
        mtf_stack_alloc_func *stack_alloc;
        mtf_stack_free_func *stack_free;
        $mtf_tcb_container *ready_container;
        mtf_lock ready_container_lock;
} mtf_scb;


/* global data */

extern unsigned mtf_stacksize;
extern mtf_scb *mtf_scb_current;
extern mtf_ecb *mtf_ecb_current[MTF_MP_MAX_VPID];


/* functions */

/* initialization and system functions */
int  mtf_init(mtf_tcb_alloc_func *tcb_alloc,
              mtf_tcb_free_func *tcb_free,
              mtf_stack_alloc_func *stack_alloc,
              mtf_stack_free_func *stack_free);
int  mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg);
void mtf_exit(int status);
int  mtf_error(char *string, int code);

/* core TCB functions */
int mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg);
int mtf_tcb_start($mtf_tcb *tcb);
int mtf_tcb_yield(void);
void mtf_tcb_exit(void);
```

```
/* malloc and free */
#define mtf_malloc(size) mtf_mp_malloc(size)
#define mtf_free(ptr) mtf_mp_free(ptr)

/* include client header file */
#include"${client_inc}"


#endif /* _${name}_h_ */
@//
@//
@//------------------------------------------------------------------------
@//
@// Generate C file
@//
@//------------------------------------------------------------------------
@perl print "Generating ${name}.c\n";
@openfile ${name}.c
/*
 * ${name}.c
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */

#include<stdio.h>
#include<stdlib.h>
#include"${name}.h"


/* constants */

unsigned mtf_stacksize = 32768;


/* data */

mtf_scb *mtf_scb_current;
mtf_ecb *mtf_ecb_current[MTF_MP_MAX_VPID];

volatile int mtf_global_exit = 0;

/* interface functions */

void mtf_yield_helper(qt_t *sp, void *old, void *blockq);
void mtf_qt_exit(void);


/*------------------------------------------------------------------------*
 * Common functions                                                        *
 *------------------------------------------------------------------------*/

/* error reporting and exiting function */

int
mtf_error(char *s, int code)
{
        printf("mtf error: %s\n", s);
        exit(code);
}
```

```
/*-----------------------------------------------------------------------*
 * Template functions                                                    *
 *-----------------------------------------------------------------------*/

/* Multiprocessor functions */

void
mtf_mp_idle_helper(qt_t *sp, void *old, void *blockq)
{
        (($mtf_tcb *) old)->sp = sp;

        /* $mtf_tcb_put(($mtf_tcb *) old, ($mtf_tcb_container *) blockq); */
        /* mtf_lock_release( *((($mtf_tcb *) old)->prev_clock) ); */
}

void
mtf_mp_idle(void)
{
        struct timespec idle_delay;

        $mtf_tcb *old, *new;

        idle_delay.tv_sec = 0;
        idle_delay.tv_nsec = 1000000;

        while (!mtf_global_exit) {

                new = $mtf_tcb_head(mtf_scb_field(ready_container));

                if (new != NULL) {
                        /* check for ready tcbs */

                        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
                        new = $mtf_tcb_get(mtf_scb_field(ready_container));
                        mtf_lock_release(mtf_scb_field(ready_container_lock));

                        if (new != NULL) {
                                /* the ready queue is not empty */
                                old = mtf_ecb_field(current);
                                mtf_tcb_set(new);
                                QT_BLOCK((qt_helper_t *) mtf_mp_idle_helper, old,
                                         (void *) NULL, new->sp);
                        } else {
                               /* mtf_lock_release(mtf_scb_field(ready_container_lock)); */

                               /* delay here */

                               nanosleep(&idle_delay, NULL);
                               /* printf("mtf_mp_idle: delayed\n"); */
                        }
                } else {
                        /* delay here */
                        nanosleep(&idle_delay, NULL);
                        /* printf("mtf_mp_idle: delayed\n"); */
                }

        }
        _exit(0);
}
```

```
void
mtf_mp_idle_start(int id)
{
        $mtf_tcb *new;

        /* allocate an execution control block */
        mtf_ecb_set((mtf_ecb *) mtf_mp_malloc(sizeof(mtf_ecb)));

        mtf_ecb_field(id) = id;

        /* allocate idle tcb */
        new = (mtf_scb_field(tcb_alloc))();

        /* initialize main tcb */
        new->next = NULL;
        new->stack = NULL;
        new->sp = NULL;

        /* set the tcb to the idle tcb for this execution control block */
        mtf_tcb_set(($mtf_tcb *) new);

        /* set the idle field for this execution control block */
        mtf_ecb_field(idle) = new;

        /* call the idle function */
        mtf_mp_idle();

        /* we should not reach this code */
        assert(0);
}


/* Core thread primitives */

int
mtf_init(mtf_tcb_alloc_func *tcb_alloc, mtf_tcb_free_func *tcb_free,
         mtf_stack_alloc_func *stack_alloc, mtf_stack_free_func *stack_free)
{
        /* initialize multiprocessor support */
        /* mtf_mp_init() must be called before lock allocation */
        mtf_mp_init();

        /* allocate a system control block */
        mtf_scb_set((mtf_scb *) malloc(sizeof(mtf_scb)));
        mtf_scb_field(tcb_alloc)   = tcb_alloc;
        mtf_scb_field(tcb_free)    = tcb_free;
        mtf_scb_field(stack_alloc) = stack_alloc;
        mtf_scb_field(stack_free)  = stack_free;
        mtf_scb_field(ready_container) =
            ($mtf_tcb_container *) malloc(sizeof($mtf_tcb_container));

        /* initialize the ready container */
        $mtf_tcb_container_init(mtf_scb_field(ready_container));

        /* initialize the ready container lock */
        mtf_lock_init(mtf_scb_field(ready_container_lock));

        /* allocate an execution control block */
        mtf_ecb_set((mtf_ecb *) malloc(sizeof(mtf_ecb)));

        return 0;
}
```

```
void
mtf_init_helper(qt_t *sp, void *old, void *blockq)
{
        /* do nothing */
}


int
mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg)
{
        /* initialize main tcb */
        main_tcb->next = NULL;
        main_tcb->stack = NULL;
        main_tcb->sp = NULL;

        mtf_tcb_init(main_tcb, main_func , main_arg);

        mtf_tcb_start(main_tcb);

        /* multiprocessor start */

        mtf_mp_start(mtf_mp_idle_start, MTF_MPCOUNT);

        /* we should not reach this code */
        assert(0);

        return 1;
}


void
mtf_create_prefix(void *u, void *t, qt_userf_t *f)
{
        mtf_tcb_set(($mtf_tcb *) t);
        ((mtf_tcb_func *)f)(u);

        /* exit with no error */
        mtf_tcb_exit();

        /* NOT REACHED */
}


int
mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg)
{
        /* allocate stack */
        tcb->stack = (mtf_scb_field(stack_alloc))(tcb->stacksize);

        /* align the stack */
        tcb->aligned_stack = MTF_QT_STKALIGN(tcb->stack, QT_STKALIGN);

        /* Get the beginning of the stack */
        tcb->sp = QT_SP (tcb->aligned_stack, mtf_stacksize - QT_STKALIGN);

        /* initialize the stack */
        tcb->sp = QT_ARGS (tcb->sp, arg, tcb, (qt_userf_t *) func,
                           mtf_create_prefix);

        return 0;
}
```

```
int
mtf_tcb_start($mtf_tcb *tcb)
{
        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
        $mtf_tcb_put(tcb, mtf_scb_field(ready_container));
        mtf_lock_release(mtf_scb_field(ready_container_lock));

        return 1;
}


void
mtf_yield_helper(qt_t *sp, void *old, void *blockq)
{
        (($mtf_tcb *) old)->sp = sp;

        $mtf_tcb_put(($mtf_tcb *) old, ($mtf_tcb_container *) blockq);
        mtf_lock_release( *((($mtf_tcb *) old)->prev_clock) );
}


int
mtf_tcb_yield(void)
{
        $mtf_tcb *old, *new;

        /* get next tcb to schedule */

        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
        new = $mtf_tcb_get(mtf_scb_field(ready_container));

        if (new != NULL) {
                /* the ready queue is not empty */
                old = mtf_ecb_field(current);
                mtf_tcb_set(new);
                old->prev_clock =
                        &(mtf_scb_field(ready_container_lock));
                QT_BLOCK((qt_helper_t *) mtf_yield_helper, old,
                       (void *) mtf_scb_field(ready_container), new->sp);
        } else {
                /* No tcbs to schedule - return to calling tcb */
                mtf_lock_release(mtf_scb_field(ready_container_lock));
        }
}


void
mtf_exit_helper(qt_t *sp, void *exiting, void *blockq)
{
        /* deallocate stack and tcb */

        assert((($mtf_tcb *) exiting) != NULL);

        if ( (($mtf_tcb *) exiting)->stack ) {
               mtf_scb_field(stack_free)( (($mtf_tcb *) exiting)->stack );
        }

        mtf_lock_release( (($mtf_tcb *) exiting)->lock );

        if ( ($mtf_tcb *) exiting) {
               mtf_scb_field(tcb_free)( ($mtf_tcb *) exiting);
        }
```

```
}


void mtf_tcb_exit(void)
{
        $mtf_tcb *old, *new;

        /* get next tcb to schedule */

        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
        new = $mtf_tcb_get(mtf_scb_field(ready_container));
        mtf_lock_release(mtf_scb_field(ready_container_lock));


        /* get idle tcb if no more ready tcbs */
        if (new == NULL) {
                new = mtf_ecb_field(idle);
        }

        if (new != NULL) {
                 old = mtf_ecb_field(current);
                 mtf_tcb_set(new);
                 QT_BLOCK((qt_helper_t *) mtf_exit_helper, old,
                         (void *) NULL, new->sp);
        } else {
                /* No tcbs to schedule */

                /* need to switch to idle thread */

                mtf_error("mtf_qt_exit: no more tcbs", 1);
                exit(1);
        }
        /* never return */
}


void
mtf_exit(int status)
{
        mtf_global_exit = 1;
        fflush(stdout);

        mtf_mp_exit(status);
        /* _exit(status); */
}


@foreach sync
@perl $block_return = "locked";
@visit block
/*
 * block helper
 */
int
${name}_helper(qt_t *sp, void *b_tcb, $cname *b_container)
{
        (($mtf_tcb *) b_tcb)->sp = sp;

        @perl for (@$list) { output("\t$_\n"); }
        mtf_lock_release( *((($mtf_tcb *) b_tcb)->prev_clock) );
}
```

```
/*
 * block current tcb and schedule a new tcb
 */
int
$name ($cname *b_container, mtf_lock *lock)
{
        $mtf_tcb *b_tcb, *s_tcb;

        /* get next tcb to schedule */
        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
        s_tcb = $mtf_tcb_get(mtf_scb_field(ready_container));
        mtf_lock_release(mtf_scb_field(ready_container_lock));

        b_tcb = mtf_tcb_ptr;

        /* get idle tcb if no more ready tcbs */
        if (s_tcb == NULL) {
                s_tcb = mtf_ecb_field(idle);
        }

        mtf_tcb_set(s_tcb);

        /* set the old tcb previous container lock */
        b_tcb->prev_clock = lock;

        /* switch to new tcb */
        QT_BLOCK((qt_helper_t *) ${name}_helper, b_tcb,
                 (void *) b_container, s_tcb->sp);

        /* re-acquire the container lock before returning to the client */
        /* pontentially this could be optimized */
        @perl if ($block_return eq "locked") {
        @perl   output("\tmtf_lock_acquire(*(b_tcb->prev_clock));\n");
        @perl }
}
@vend

@visit unblock
/*
 * unblock a tcb
 */
int
$name ($cname *b_container)
{
        $mtf_tcb *b_tcb;
        /* get tcb from container */
        @perl for (@$list) { output("\t$_\n"); }

        assert(b_tcb != NULL);

        /* put tcb on ready container */
        if (b_tcb != NULL) {
                mtf_lock_acquire(mtf_scb_field(ready_container_lock));
                $mtf_tcb_put(b_tcb, mtf_scb_field(ready_container));
                mtf_lock_release(mtf_scb_field(ready_container_lock));
        }

}
@vend
@end
```

A.2.5 mtf_pt.tpl

```
@//
@// mtf_pt.h
@//
@// template for Pthreads (pt) target
@//
@//------------------------------------------------------------------------
@//
@// define global variables
@//
@//------------------------------------------------------------------------
@visit sched
@perl $mtf_tcb_container = $cname;
@perl $mtf_tcb_container_init = $init;
@perl $mtf_tcb_put  = $put;
@perl $mtf_tcb_get  = $get;
@vend
@//
@perl $user = $ENV{"USER"};
@//
@//
@//------------------------------------------------------------------------
@//
@// Generate header file
@//
@//------------------------------------------------------------------------
@perl print "Generating ${name}.h\n";
@openfile ${name}.h
/*
 * ${name}.h
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */
#ifndef _${name}_h_
#define _${name}_h_

/* include files */

/* hack for OSF1 */
#ifdef __osf__
#define __C_ASM_H
#endif

#include <assert.h>
#include <pthread.h>
#include <sched.h>

#ifdef __osf__
#include <setjmp.h>
#endif

/* macros */

@visit tcb
@perl $mtf_tcb = $name;
#define mtf_scb_ptr mtf_scb_current
#define mtf_scb_field(x) mtf_scb_ptr->x
#define mtf_scb_set(newscb) mtf_scb_ptr = newscb

/* not used */
```

```
#define mtf_ecb_ptr (mtf_tcb_field(ecb))
#define mtf_ecb_field(x) mtf_ecb_ptr->x
#define mtf_ecb_set(newecb) mtf_ecb_ptr = newecb

#define mtf_tcb_ptr (($mtf_tcb *) pthread_getspecific(mtf_tcb_key))
#define mtf_tcb_field(x) (mtf_tcb_ptr)->x
#define mtf_tcb_set(newtcb) pthread_setspecific(mtf_tcb_key, (void *) newtcb)

#define mtf_tcb_current() mtf_tcb_ptr
@vend

/* datatypes */

/* locks */
#define mtf_lock                pthread_mutex_t
#define mtf_lock_init(lock)     pthread_mutex_init(&lock, NULL)
#define mtf_lock_free(lock)     pthread_mutex_destroy(&lock)
#define mtf_lock_acquire(lock)  pthread_mutex_lock(&lock)
#define mtf_lock_release(lock)  pthread_mutex_unlock(&lock)


/* thread control block (tcb) datatypes */

/* tcb function pointer */
typedef void * (mtf_tcb_func)(void *arg);

typedef void * (mtf_pthread_func)(void *arg);

/* thread control block */
@visit tcb
@perl $mtf_tcb = $name;
typedef struct $name {
        /* client supplied tcb fields */
        @perl for (@$list) { output("\t$_\n"); }

        /* target fields */
        mtf_tcb_func *func;
        void *arg;
        pthread_t pthread_handle;
        pthread_mutex_t mutex;
        pthread_cond_t  cv;
        #ifdef __osf__
        jmp_buf prefix_state;
        #endif
} $name;
@vend

/* thread control block containers */
@visit tcb_container
typedef struct $name {
        @perl for (@$list) { output("\t$_\n"); }
} $name;
@vend

/* execution control block */
typedef struct mtf_ecb
{
        $mtf_tcb *current;
        $mtf_tcb *idle;
        $mtf_tcb *exitfunc;
        $mtf_tcb *exiting;
} mtf_ecb;
```

```
typedef $mtf_tcb * (mtf_tcb_alloc_func)(void);
typedef void (mtf_tcb_free_func)($mtf_tcb *);
typedef void * (mtf_stack_alloc_func)(size_t);
typedef void (mtf_stack_free_func)(void *);

typedef struct mtf_scb
{
        mtf_tcb_alloc_func *tcb_alloc;
        mtf_tcb_free_func *tcb_free;
        mtf_stack_alloc_func *stack_alloc;
        mtf_stack_free_func *stack_free;
        $mtf_tcb_container *ready_container;
} mtf_scb;


/* global data */

extern unsigned mtf_stacksize;
extern pthread_key_t mtf_tcb_key;


/* functions */

/* initialization and system functions */
int  mtf_init(mtf_tcb_alloc_func *tcb_alloc,
              mtf_tcb_free_func *tcb_free,
              mtf_stack_alloc_func *stack_alloc,
              mtf_stack_free_func *stack_free);
int  mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg);
void mtf_exit(int status);
int  mtf_error(char *string, int code);

/* core TCB functions */
int mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg);
int mtf_tcb_start($mtf_tcb *tcb);
int mtf_tcb_yield(void);
void mtf_tcb_exit(void);

/* malloc and free */
#define mtf_malloc(size) malloc(size)
#define mtf_free(ptr) free(ptr)

/* include client header file */
#include"${client_inc}"


#endif /* _${name}_h_ */
@//
@//
@//-----------------------------------------------------------------------
@//
@// Generate C file
@//
@//-----------------------------------------------------------------------
@perl print "Generating ${name}.c\n";
@openfile ${name}.c
/*
 * ${name}.c
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
```

```
 * Copyright (C) 1998 by Gregory D. Benson
 */

#include<stdio.h>
#include<stdlib.h>
#include"${name}.h"


/* constants */

unsigned mtf_stacksize = 32768;


/* data */

mtf_scb *mtf_scb_current;
pthread_key_t mtf_tcb_key;
pthread_attr_t mtf_pthread_attr_default;




/* interface functions */


/* error reporting and exiting function */

int
mtf_error(char *s, int code)
{
        printf("mtf error: %s\n", s);
        exit(code);
}


/* template functions */

int
mtf_init(mtf_tcb_alloc_func *tcb_alloc, mtf_tcb_free_func *tcb_free,
         mtf_stack_alloc_func *stack_alloc, mtf_stack_free_func *stack_free)
{
        pthread_attr_init(&mtf_pthread_attr_default);
        pthread_attr_setdetachstate(&mtf_pthread_attr_default,
                                    PTHREAD_CREATE_DETACHED);

        pthread_key_create(&mtf_tcb_key, NULL);
        pthread_detach(pthread_self());

        /* set the concurrency level */
        #ifndef __linux__
        pthread_setconcurrency(MTF_MPCOUNT);
        #endif

        /* allocate a system control block */
        mtf_scb_set((mtf_scb *) malloc(sizeof(mtf_scb)));
        mtf_scb_field(tcb_alloc)   = tcb_alloc;
        mtf_scb_field(tcb_free)    = tcb_free;
        mtf_scb_field(stack_alloc) = stack_alloc;
        mtf_scb_field(stack_free)  = stack_free;
        mtf_scb_field(ready_container) =
            ($mtf_tcb_container *) malloc(sizeof($mtf_tcb_container));

        /* initialize the ready container */
```

```
        $mtf_tcb_container_init(mtf_scb_field(ready_container));

        return 0;
}


int
mtf_start($mtf_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg)
{
        /* initialize main tcb */
        main_tcb->next = NULL;

        mtf_tcb_init(main_tcb, main_func, main_arg);
        mtf_tcb_start(main_tcb);
        pthread_exit(NULL);

        /* not reached */
        return 1;
}


int
mtf_tcb_init($mtf_tcb *tcb, mtf_tcb_func *func, void *arg)
{
        tcb->func = func;
        tcb->arg  = arg;
        pthread_mutex_init(&(tcb->mutex), NULL);
        pthread_cond_init(&(tcb->cv), NULL);

        return 1;
}


void
mtf_create_prefix($mtf_tcb *tcb)
{
        mtf_tcb_set(tcb);

        #ifdef __osf__
                if (_setjmp(tcb->prefix_state) == 0) {
                        (tcb->func)(tcb->arg);
                }
                (mtf_scb_field(tcb_free))(tcb);
        #else
               (tcb->func)(tcb->arg);
               mtf_tcb_exit();
        #endif
}


int
mtf_tcb_start($mtf_tcb *tcb)
{
        int rv;

        rv = pthread_create(&(tcb->pthread_handle), &mtf_pthread_attr_default,
                            (mtf_pthread_func *) mtf_create_prefix, (void *) tcb);

        if (rv != 0) {
           printf("mtf_tcb_start: rv = %d\n", rv);
           perror("mtf_tcb_start");
        }
        assert (rv == 0);
```

```
        return 1;
}

int
mtf_tcb_yield(void)
{
        sched_yield();
}


void mtf_tcb_exit(void)
{
        $mtf_tcb *current;

        current = mtf_tcb_ptr;

        mtf_lock_release( (($mtf_tcb *) current)->lock );

        #ifdef __osf__
                _longjmp(current->prefix_state, 1);
        #else
                (mtf_scb_field(tcb_free))(current);
                pthread_exit((void *) NULL);
        #endif
}


void
mtf_exit(int status)
{
        fflush(stdout);
        exit(status);
}


@foreach sync
@visit block
/*
 * block current tcb and schedule a new tcb
 */
int
$name ($cname *b_container, mtf_lock *lock)
{
        $mtf_tcb *b_tcb;

        /* get current thread */
        b_tcb = mtf_tcb_ptr;

        @perl for (@$list) { output("\t$_\n"); }

        pthread_cond_wait(&(b_tcb->cv), lock);
}
@vend

@visit unblock
/*
 * unblock a tcb
 */
int
$name ($cname *b_container)
{
        $mtf_tcb *b_tcb;
```

```
        /* get tcb from container */
        @perl for (@$list) { output("\t$_\n"); }

        pthread_cond_signal(&(b_tcb->cv));
}
@vend
@end
```

Appendix B

MutexThreads Source Code

B.1 Client Code

B.1.1 mutexgen.mtf

```
#
# mutexgen.mtf
#
# Mezcla client description file for MutexThreads

# Preamble

set name mutexgen
set client_inc mutexclient.h

# Data Descriptions

data tcb mutex_tcb {
        struct mutex_tcb *next;
        unsigned stacksize;
        mtf_lock lock;
}

data tcb_container mutex_container {
        struct mutex_tcb *head;
        struct mutex_tcb *tail;
}


# Code

code sched mutex_container {
        set init mutex_container_init
        set put  mutex_tcb_put
        set get  mutex_tcb_get
        set head mutex_tcb_head
}

codel sync mutex_container {
        block mutex_block {
                mutex_tcb_put(b_tcb, b_container);
```

```
        }
        unblock mutex_unblock {
                b_tcb = mutex_tcb_get(b_container);
        }
}
```

B.1.2 mutexclient.h

```
/*
 * mutexclient.h
 *
 * mtf - The Mezcla Thread Framework
 * Copyright (C) 1999 by Gregory D. Benson
 */


/* this is needed to allow unoptimized compilation */
#ifndef MTF_INLINE
#define MTF_INLINE extern __inline__
#endif


/* TCB container functions */

MTF_INLINE int
mutex_container_init(mutex_container *container)
{
        container->head = NULL;
        container->tail = NULL;
}


MTF_INLINE int
mutex_tcb_put(mutex_tcb *tcb, mutex_container *container)
{
        tcb->next = 0;
        if (container->tail == 0) {
                container->head = tcb;
        } else {
                container->tail->next = tcb;
        }
        container->tail = tcb;
        return 1;
}

MTF_INLINE mutex_tcb *
mutex_tcb_get(mutex_container *container)
{
        mutex_tcb *tcb;

        if ((tcb = container->head) != NULL &&
            ((container->head = tcb->next) == 0)) {
                container->tail = 0;
        }

        return tcb;
}

MTF_INLINE mutex_tcb *
mutex_tcb_head(mutex_container *container)
{
```

```
        return container->head;
}
```

B.1.3 mutexthreads.h

```
/*
 * mtf - The Mezcla Thread Framework
 * Copyright (C) 1999 by Gregory D. Benson
 */

/*
 * mutexthreads.h - a simple thread system based on mutex variables
 */

#include "mutexgen.h"

/* datatypes */

typedef struct mutex {
        int value;
        mtf_lock lock;
        mutex_container container;
} mutex;


/* functions */

void mutex_start(mtf_tcb_func *func, void *arg);

void mutex_create(mtf_tcb_func *func, void *arg);
void mutex_exit(void);

void mutex_init(mutex *m);
void mutex_destroy(mutex *m);
void mutex_lock(mutex *m);
void mutex_unlock(mutex *m);
```

B.1.4 mutexthreads.c

```
/*
 * mtf - The Mezcla Thread Framework
 * Copyright (C) 1999 by Gregory D. Benson
 */

/*
 * mutexthreads.c - a simple thread system based on mutex variables
 */


#include <stdio.h>
#include <stdlib.h>
#include "mutexthreads.h"

/* this is needed to allow unoptimized compilation */
#undef MTF_INLINE
#define MTF_INLINE
#include "mutexclient.h"


/* variables */
```

```
static mutex_tcount = 0;
static mtf_lock mutex_tcount_lock;


/* functions */

mutex_tcb *
mutex_tcb_alloc()
{
        return (mutex_tcb *) mtf_malloc(sizeof(mutex_tcb));
}

void
mutex_tcb_free(mutex_tcb *tcb)
{
        mtf_free((void *) tcb);
}

void *
mutex_stack_alloc(size_t size)
{
        return (void *) mtf_malloc(size);
}

void
mutex_stack_free(void *stack)
{
        mtf_free(stack);
}


void
mutex_start(mtf_tcb_func func, void *arg)
{
        mutex_tcb *tcb_main;

        mtf_init(mutex_tcb_alloc, mutex_tcb_free,
                 mutex_stack_alloc, mutex_stack_free);

        mtf_lock_init(mutex_tcount_lock);
        mutex_tcount = 1;

        /* printf("mtf_stacksize   = %d\n", mtf_stacksize); */
        /* printf("sizeof(mtf_tcb) = %d\n", sizeof(mutex_tcb)); */
        /* tcb allocation should be a callback */
        tcb_main = (mutex_tcb *) mtf_malloc(sizeof(mutex_tcb));
        tcb_main->stacksize = mtf_stacksize;

        /* printf("calling mtf_init(%X)\n", tcb_main); */
        mtf_start(tcb_main, (mtf_tcb_func *) func, arg);
}


void
mutex_create(mtf_tcb_func *func, void *arg)
{
        mutex_tcb *new;

        new = mutex_tcb_alloc();
        new->stacksize = mtf_stacksize;

        mtf_lock_init(new->lock);
```

```
        mtf_tcb_init(new, (mtf_tcb_func *) func, arg);

        /* set global thread count */
        mtf_lock_acquire(mutex_tcount_lock);
        mutex_tcount++;
        mtf_lock_release(mutex_tcount_lock);

        mtf_tcb_start(new);
}


void
mutex_exit(void)
{
        mutex_tcb *p;
        int exit = 0;
        p = mtf_tcb_ptr;

        mtf_lock_acquire(mutex_tcount_lock);
        mutex_tcount--;
        if (mutex_tcount == 1) {
                exit = 1;
        }
        mtf_lock_release(mutex_tcount_lock);

        /* if this is the last thread to exit then exit the process */
        if (exit == 0) {
                mtf_lock_acquire(p->lock);
                mtf_tcb_exit();
        } else {
                mtf_exit(0);
        }
}


void
mutex_init(mutex *m)
{
        m->value = 0;
        mtf_lock_init(m->lock);
        mutex_container_init(&(m->container));
}


void
mutex_destroy(mutex *m)
{
        m->value = 0;
        mtf_lock_free(m->lock);
}


void
mutex_lock(mutex *m)
{
        mtf_lock_acquire(m->lock);

        if ((m->value) == 0) {
                m->value = 1;
        } else {
                mutex_block(&(m->container), &(m->lock));
        }
```

```
        mtf_lock_release(m->lock);
}


void
mutex_unlock(mutex *m)
{
        mtf_lock_acquire(m->lock);

        if (mutex_tcb_head(&(m->container)) != ((mutex_tcb *) NULL)) {
                mutex_unblock(&(m->container));
        } else {
                m->value = 0;
        }

        mtf_lock_release(m->lock);
}
```

B.2 Generated Code

B.2.1 Preswitch Multiprocessor

B.2.1.1 mutexgen.h

```
/*
 * mutexgen.h
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1999 by Gregory D. Benson
 */

#ifndef _mutexgen_h_
#define _mutexgen_h_

/* include files */
#include <assert.h>
#include <mtf/qt.h>
#include <mtf/system/mp.h>

/* macros */

#define mtf_scb_ptr mtf_scb_current
#define mtf_scb_field(x) mtf_scb_ptr->x
#define mtf_scb_set(newscb) mtf_scb_ptr = newscb

#define mtf_ecb_ptr mtf_ecb_current[MTF_MP_GET_VPID]
#define mtf_ecb_field(x) mtf_ecb_ptr->x
#define mtf_ecb_set(newecb) mtf_ecb_ptr = newecb

#define mtf_tcb_ptr mtf_ecb_field(current)
#define mtf_tcb_field(x) (mtf_tcb_ptr)->x
#define mtf_tcb_set(newtcb) mtf_tcb_ptr = newtcb

#define mtf_tcb_current() mtf_tcb_ptr
```

```
#define MTF_QT_STKALIGN(sp, alignment) \
  ((void *)((((qt_word_t)(sp)) + (alignment) - 1) & ~((alignment)-1)))


/* datatypes */

/* locks */
#define mtf_lock                mtf_mp_lock_t
#define mtf_lock_init(lock)     mtf_mp_lock_init(lock)
#define mtf_lock_free(lock)     mtf_mp_lock_free(lock)
#define mtf_lock_acquire(lock)  mtf_mp_lock_acquire(lock)
#define mtf_lock_release(lock)  mtf_mp_lock_release(lock)

/* thread control block (tcb) datatypes */

/* tcb function pointer */
typedef void * (mtf_tcb_func)(void *arg);

/* thread control block */
typedef struct mutex_tcb {
        /* client supplied tcb fields */
struct mutex_tcb *next;
unsigned stacksize;
mtf_lock lock;

        /* target fields */
        void *stack;
        void *aligned_stack;
        qt_t *sp;
        mtf_lock *prev_lock;
} mutex_tcb;

/* thread control block containers */
typedef struct mutex_container {
struct mutex_tcb *head;
struct mutex_tcb *tail;
} mutex_container;

/* execution control block */
typedef struct mtf_ecb
{
        int      id;
        mutex_tcb *current;
        mutex_tcb *idle;
        /* mutex_tcb *exitfunc; */
        /* mutex_tcb *exiting; */
} mtf_ecb;


/* system control block */
typedef mutex_tcb * (mtf_tcb_alloc_func)(void);
typedef void (mtf_tcb_free_func)(mutex_tcb *);
typedef void * (mtf_stack_alloc_func)(size_t);
typedef void (mtf_stack_free_func)(void *);

typedef struct mtf_scb
{
        mtf_tcb_alloc_func *tcb_alloc;
        mtf_tcb_free_func *tcb_free;
        mtf_stack_alloc_func *stack_alloc;
        mtf_stack_free_func *stack_free;
        mutex_container *ready_container;
        mtf_lock ready_container_lock;
```

```
} mtf_scb;


/* global data */

extern unsigned mtf_stacksize;
extern mtf_scb *mtf_scb_current;
extern mtf_ecb *mtf_ecb_current[MTF_MP_MAX_VPID];


/* functions */

/* initialization and system functions */
int  mtf_init(mtf_tcb_alloc_func *tcb_alloc,
              mtf_tcb_free_func *tcb_free,
              mtf_stack_alloc_func *stack_alloc,
              mtf_stack_free_func *stack_free);
int  mtf_start(mutex_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg);
void mtf_exit(int status);
int  mtf_error(char *string, int code);

/* core TCB functions */
int mtf_tcb_init(mutex_tcb *tcb, mtf_tcb_func *func, void *arg);
int mtf_tcb_start(mutex_tcb *tcb);
int mtf_tcb_yield(void);
void mtf_tcb_exit(void);

/* malloc and free */
#define mtf_malloc(size) mtf_mp_malloc(size)
#define mtf_free(ptr) mtf_mp_free(ptr)

/* include client header file */
#include"mutexclient.h"


#endif /* _mutexgen_h_ */
```

B.2.1.2 mutexgen.c

```
/*
 * mutexgen.c
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */

#include<stdio.h>
#include<stdlib.h>
#include"mutexgen.h"


/* constants */

unsigned mtf_stacksize = 32768;


/* data */

mtf_scb *mtf_scb_current;
mtf_ecb *mtf_ecb_current[MTF_MP_MAX_VPID];
```

```
volatile int mtf_global_exit = 0;

/* interface functions */

void mtf_yield_helper(qt_t *sp, void *old, void *blockq);
void mtf_qt_exit(void);


/*------------------------------------------------------------------------*
 * Common functions                                                       *
 *------------------------------------------------------------------------*/

/* error reporting and exiting function */

int
mtf_error(char *s, int code)
{
        printf("mtf error: %s\n", s);
        exit(code);
}


/*------------------------------------------------------------------------*
 * Template functions                                                     *
 *------------------------------------------------------------------------*/

/* Multiprocessor functions */

void
mtf_mp_idle_helper(qt_t *sp, void *old, void *blockq)
{
        ((mutex_tcb *) old)->sp = sp;

        /* mutex_tcb_put((mutex_tcb *) old, (mutex_container *) blockq); */
        /* mtf_lock_release( *(((mutex_tcb *) old)->prev_lock) ); */
}

void
mtf_mp_idle(void)
{
        struct timespec idle_delay;

        mutex_tcb *old, *new;

        idle_delay.tv_sec = 0;
        idle_delay.tv_nsec = 1000000;

        while (!mtf_global_exit) {

                new = mutex_tcb_head(mtf_scb_field(ready_container));

                if (new != NULL) {
                        /* check for ready tcbs */

                        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
                        new = mutex_tcb_get(mtf_scb_field(ready_container));
                        mtf_lock_release(mtf_scb_field(ready_container_lock));

                        if (new != NULL) {
                                /* the ready queue is not empty */
                                old = mtf_ecb_field(current);
                                mtf_tcb_set(new);
```

```
                                 QT_BLOCK((qt_helper_t *) mtf_mp_idle_helper, old,
                                          (void *) NULL, new->sp);
                        } else {
                                /* mtf_lock_release(mtf_scb_field(ready_container_lock)); */

                                /* delay here */

                                nanosleep(&idle_delay, NULL);
                                /* printf("mtf_mp_idle: delayed\n"); */
                        }
                } else {
                        /* delay here */
                        nanosleep(&idle_delay, NULL);
                        /* printf("mtf_mp_idle: delayed\n"); */
                }

        }
        _exit(0);
}


void
mtf_mp_idle_start(int id)
{
        mutex_tcb *new;

        /* allocate an execution control block */
        mtf_ecb_set((mtf_ecb *) mtf_mp_malloc(sizeof(mtf_ecb)));

        mtf_ecb_field(id) = id;

        /* allocate idle tcb */
        new = (mtf_scb_field(tcb_alloc))();

        /* initialize main tcb */
        new->next = NULL;
        new->stack = NULL;
        new->sp = NULL;

        /* set the tcb to the idle tcb for this execution control block */
        mtf_tcb_set((mutex_tcb *) new);

        /* set the idle field for this execution control block */
        mtf_ecb_field(idle) = new;

        /* call the idle function */
        mtf_mp_idle();

        /* we should not reach this code */
        assert(0);
}


/* Core thread primitives */

int
mtf_init(mtf_tcb_alloc_func *tcb_alloc, mtf_tcb_free_func *tcb_free,
         mtf_stack_alloc_func *stack_alloc, mtf_stack_free_func *stack_free)
{
        /* initialize multiprocessor support */
        /* mtf_mp_init() must be called before lock allocation */
        mtf_mp_init();
```

```
        /* allocate a system control block */
        mtf_scb_set((mtf_scb *) malloc(sizeof(mtf_scb)));
        mtf_scb_field(tcb_alloc)   = tcb_alloc;
        mtf_scb_field(tcb_free)    = tcb_free;
        mtf_scb_field(stack_alloc) = stack_alloc;
        mtf_scb_field(stack_free)  = stack_free;
        mtf_scb_field(ready_container) =
            (mutex_container *) malloc(sizeof(mutex_container));

        /* initialize the ready container */
        mutex_container_init(mtf_scb_field(ready_container));

        /* initialize the ready container lock */
        mtf_lock_init(mtf_scb_field(ready_container_lock));

        /* allocate an execution control block */
        mtf_ecb_set((mtf_ecb *) malloc(sizeof(mtf_ecb)));

        return 0;
}


void
mtf_init_helper(qt_t *sp, void *old, void *blockq)
{
        /* do nothing */
}


int
mtf_start(mutex_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg)
{
        /* initialize main tcb */
        main_tcb->next = NULL;
        main_tcb->stack = NULL;
        main_tcb->sp = NULL;

        mtf_tcb_init(main_tcb, main_func , main_arg);

        mtf_tcb_start(main_tcb);

        /* multiprocessor start */

        mtf_mp_start(mtf_mp_idle_start, MTF_MPCOUNT);

        /* we should not reach this code */
        assert(0);

        return 1;
}


void
mtf_create_prefix(void *u, void *t, qt_userf_t *f)
{
        mtf_tcb_set((mutex_tcb *) t);
        ((mtf_tcb_func *)f)(u);

        /* exit with no error */
        mtf_tcb_exit();

        /* NOT REACHED */
}
```

```
int
mtf_tcb_init(mutex_tcb *tcb, mtf_tcb_func *func, void *arg)
{
        /* allocate stack */
        tcb->stack = (mtf_scb_field(stack_alloc))(tcb->stacksize);

        /* align the stack */
        tcb->aligned_stack = MTF_QT_STKALIGN(tcb->stack, QT_STKALIGN);

        /* Get the beginning of the stack */
        tcb->sp = QT_SP (tcb->aligned_stack, mtf_stacksize - QT_STKALIGN);

        /* initialize the stack */
        tcb->sp = QT_ARGS (tcb->sp, arg, tcb, (qt_userf_t *) func,
                           mtf_create_prefix);

        return 0;
}


int
mtf_tcb_start(mutex_tcb *tcb)
{
        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
        mutex_tcb_put(tcb, mtf_scb_field(ready_container));
        mtf_lock_release(mtf_scb_field(ready_container_lock));

        return 1;
}


void
mtf_yield_helper(qt_t *sp, void *old, void *blockq)
{
        ((mutex_tcb *) old)->sp = sp;

        mutex_tcb_put((mutex_tcb *) old, (mutex_container *) blockq);
        mtf_lock_release( *(((mutex_tcb *) old)->prev_lock) );
}


int
mtf_tcb_yield(void)
{
        mutex_tcb *old, *new;

        /* get next tcb to schedule */

        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
        new = mutex_tcb_get(mtf_scb_field(ready_container));

        if (new != NULL) {
                /* the ready queue is not empty */
                old = mtf_ecb_field(current);
                mtf_tcb_set(new);
                old->prev_lock =
                        &(mtf_scb_field(ready_container_lock));
                QT_BLOCK((qt_helper_t *) mtf_yield_helper, old,
                       (void *) mtf_scb_field(ready_container), new->sp);
        } else {
                /* No tcbs to schedule - return to calling tcb */
```

```
                 mtf_lock_release(mtf_scb_field(ready_container_lock));
        }
}


void
mtf_exit_helper(qt_t *sp, void *exiting, void *blockq)
{
        /* deallocate stack and tcb */

        assert(((mutex_tcb *) exiting) != NULL);

        if ( ((mutex_tcb *) exiting)->stack ) {
                mtf_scb_field(stack_free)( ((mutex_tcb *) exiting)->stack );
        }

        mtf_lock_release( ((mutex_tcb *) exiting)->lock );

        if ( (mutex_tcb *) exiting) {
                mtf_scb_field(tcb_free)( (mutex_tcb *) exiting);
        }
}


void mtf_tcb_exit(void)
{
        mutex_tcb *old, *new;

        /* get next tcb to schedule */

        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
        new = mutex_tcb_get(mtf_scb_field(ready_container));
        mtf_lock_release(mtf_scb_field(ready_container_lock));


        /* get idle tcb if no more ready tcbs */
        if (new == NULL) {
                new = mtf_ecb_field(idle);
        }

        if (new != NULL) {
                 old = mtf_ecb_field(current);
                 mtf_tcb_set(new);
                 QT_BLOCK((qt_helper_t *) mtf_exit_helper, old,
                         (void *) NULL, new->sp);
        } else {
                /* No tcbs to schedule */

                /* need to switch to idle thread */

                mtf_error("mtf_qt_exit: no more tcbs", 1);
                exit(1);
        }
        /* never return */
}


void
mtf_exit(int status)
{
        mtf_global_exit = 1;
        fflush(stdout);
```

```
        mtf_mp_exit(status);
        /* _exit(status); */
}


/*
 * block helper
 */
int
mutex_block_helper(qt_t *sp, void *b_tcb, mutex_container *b_container)
{
        ((mutex_tcb *) b_tcb)->sp = sp;

mutex_tcb_put(b_tcb, b_container);
        mtf_lock_release( *(((mutex_tcb *) b_tcb)->prev_lock) );
}


/*
 * block current tcb and schedule a new tcb
 */
int
mutex_block (mutex_container *b_container, mtf_lock *lock)
{
        mutex_tcb *b_tcb, *s_tcb;

        /* get next tcb to schedule */
        mtf_lock_acquire(mtf_scb_field(ready_container_lock));
        s_tcb = mutex_tcb_get(mtf_scb_field(ready_container));
        mtf_lock_release(mtf_scb_field(ready_container_lock));

        b_tcb = mtf_tcb_ptr;

        /* get idle tcb if no more ready tcbs */
        if (s_tcb == NULL) {
                s_tcb = mtf_ecb_field(idle);
        }

        mtf_tcb_set(s_tcb);

        /* set the old tcb previous container lock */
        b_tcb->prev_lock = lock;

        /* switch to new tcb */
        QT_BLOCK((qt_helper_t *) mutex_block_helper, b_tcb,
                 (void *) b_container, s_tcb->sp);

        /* re-acquire the container lock before returning to the client */
        /* pontentially this could be optimized */
mtf_lock_acquire(*(b_tcb->prev_lock));
}

/*
 * unblock a tcb
 */
int
mutex_unblock (mutex_container *b_container)
{
        mutex_tcb *b_tcb;
        /* get tcb from container */
b_tcb = mutex_tcb_get(b_container);

        assert(b_tcb != NULL);
```

```
        /* put tcb on ready container */
        if (b_tcb != NULL) {
                mtf_lock_acquire(mtf_scb_field(ready_container_lock));
                mutex_tcb_put(b_tcb, mtf_scb_field(ready_container));
                mtf_lock_release(mtf_scb_field(ready_container_lock));
        }

}
```

B.2.2 Pthreads

B.2.2.1 mutexgen.h

```
/*
 * mutexgen.h
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */
#ifndef _mutexgen_h_
#define _mutexgen_h_

/* include files */

/* hack for OSF1 */
#ifdef __osf__
#define __C_ASM_H
#endif

#include <assert.h>
#include <pthread.h>
#include <sched.h>

#ifdef __osf__
#include <setjmp.h>
#endif

/* macros */

#define mtf_scb_ptr mtf_scb_current
#define mtf_scb_field(x) mtf_scb_ptr->x
#define mtf_scb_set(newscb) mtf_scb_ptr = newscb

/* not used */
#define mtf_ecb_ptr (mtf_tcb_field(ecb))
#define mtf_ecb_field(x) mtf_ecb_ptr->x
#define mtf_ecb_set(newecb) mtf_ecb_ptr = newecb

#define mtf_tcb_ptr ((mutex_tcb *) pthread_getspecific(mtf_tcb_key))
#define mtf_tcb_field(x) (mtf_tcb_ptr)->x
#define mtf_tcb_set(newtcb) pthread_setspecific(mtf_tcb_key, (void *) newtcb)

#define mtf_tcb_current() mtf_tcb_ptr

/* datatypes */

/* locks */
#define mtf_lock                pthread_mutex_t
```

```
#define mtf_lock_init(lock)     pthread_mutex_init(&lock, NULL)
#define mtf_lock_free(lock)     pthread_mutex_destroy(&lock)
#define mtf_lock_acquire(lock)  pthread_mutex_lock(&lock)
#define mtf_lock_release(lock)  pthread_mutex_unlock(&lock)


/* thread control block (tcb) datatypes */

/* tcb function pointer */
typedef void * (mtf_tcb_func)(void *arg);

typedef void * (mtf_pthread_func)(void *arg);

/* thread control block */
typedef struct mutex_tcb {
        /* client supplied tcb fields */
struct mutex_tcb *next;
unsigned stacksize;
mtf_lock lock;

        /* target fields */
        mtf_tcb_func *func;
        void *arg;
        pthread_t pthread_handle;
        pthread_mutex_t mutex;
        pthread_cond_t  cv;
        #ifdef __osf__
        jmp_buf prefix_state;
        #endif
} mutex_tcb;

/* thread control block containers */
typedef struct mutex_container {
struct mutex_tcb *head;
struct mutex_tcb *tail;
} mutex_container;

/* execution control block */
typedef struct mtf_ecb
{
        mutex_tcb *current;
        mutex_tcb *idle;
        mutex_tcb *exitfunc;
        mutex_tcb *exiting;
} mtf_ecb;


typedef mutex_tcb * (mtf_tcb_alloc_func)(void);
typedef void (mtf_tcb_free_func)(mutex_tcb *);
typedef void * (mtf_stack_alloc_func)(size_t);
typedef void (mtf_stack_free_func)(void *);

typedef struct mtf_scb
{
        mtf_tcb_alloc_func *tcb_alloc;
        mtf_tcb_free_func *tcb_free;
        mtf_stack_alloc_func *stack_alloc;
        mtf_stack_free_func *stack_free;
        mutex_container *ready_container;
} mtf_scb;


/* global data */
```

```
extern unsigned mtf_stacksize;
extern pthread_key_t mtf_tcb_key;


/* functions */

/* initialization and system functions */
int  mtf_init(mtf_tcb_alloc_func *tcb_alloc,
              mtf_tcb_free_func *tcb_free,
              mtf_stack_alloc_func *stack_alloc,
              mtf_stack_free_func *stack_free);
int  mtf_start(mutex_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg);
void mtf_exit(int status);
int  mtf_error(char *string, int code);

/* core TCB functions */
int mtf_tcb_init(mutex_tcb *tcb, mtf_tcb_func *func, void *arg);
int mtf_tcb_start(mutex_tcb *tcb);
int mtf_tcb_yield(void);
void mtf_tcb_exit(void);

/* malloc and free */
#define mtf_malloc(size) malloc(size)
#define mtf_free(ptr) free(ptr)

/* include client header file */
#include"mutexclient.h"


#endif /* _mutexgen_h_ */
```

B.2.2.2 mutexgen.c

```
/*
 * mutexgen.c
 *
 * This file is automatically generated by tgen
 *
 * mtf - the Mezcla Thread Framework
 * Copyright (C) 1998 by Gregory D. Benson
 */

#include<stdio.h>
#include<stdlib.h>
#include"mutexgen.h"


/* constants */

unsigned mtf_stacksize = 32768;


/* data */

mtf_scb *mtf_scb_current;
pthread_key_t mtf_tcb_key;
pthread_attr_t mtf_pthread_attr_default;
```

```
/* interface functions */


/* error reporting and exiting function */

int
mtf_error(char *s, int code)
{
        printf("mtf error: %s\n", s);
        exit(code);
}


/* template functions */

int
mtf_init(mtf_tcb_alloc_func *tcb_alloc, mtf_tcb_free_func *tcb_free,
         mtf_stack_alloc_func *stack_alloc, mtf_stack_free_func *stack_free)
{
        pthread_attr_init(&mtf_pthread_attr_default);
        pthread_attr_setdetachstate(&mtf_pthread_attr_default,
                                    PTHREAD_CREATE_DETACHED);

        pthread_key_create(&mtf_tcb_key, NULL);
        pthread_detach(pthread_self());

        /* set the concurrency level */
        #ifndef __linux__
        pthread_setconcurrency(MTF_MPCOUNT);
        #endif

        /* allocate a system control block */
        mtf_scb_set((mtf_scb *) malloc(sizeof(mtf_scb)));
        mtf_scb_field(tcb_alloc)   = tcb_alloc;
        mtf_scb_field(tcb_free)    = tcb_free;
        mtf_scb_field(stack_alloc) = stack_alloc;
        mtf_scb_field(stack_free)  = stack_free;
        mtf_scb_field(ready_container) =
            (mutex_container *) malloc(sizeof(mutex_container));

        /* initialize the ready container */
        mutex_container_init(mtf_scb_field(ready_container));

        return 0;
}


int
mtf_start(mutex_tcb *main_tcb, mtf_tcb_func *main_func, void *main_arg)
{
        /* initialize main tcb */
        main_tcb->next = NULL;

        mtf_tcb_init(main_tcb, main_func, main_arg);
        mtf_tcb_start(main_tcb);
        pthread_exit(NULL);

        /* not reached */
        return 1;
}


int
```

```
mtf_tcb_init(mutex_tcb *tcb, mtf_tcb_func *func, void *arg)
{
        tcb->func = func;
        tcb->arg  = arg;
        pthread_mutex_init(&(tcb->mutex), NULL);
        pthread_cond_init(&(tcb->cv), NULL);

        return 1;
}


void
mtf_create_prefix(mutex_tcb *tcb)
{
        mtf_tcb_set(tcb);

        #ifdef __osf__
                if (_setjmp(tcb->prefix_state) == 0) {
                        (tcb->func)(tcb->arg);
                }
                (mtf_scb_field(tcb_free))(tcb);
        #else
               (tcb->func)(tcb->arg);
               mtf_tcb_exit();
        #endif
}


int
mtf_tcb_start(mutex_tcb *tcb)
{
        int rv;

        rv = pthread_create(&(tcb->pthread_handle), &mtf_pthread_attr_default,
                            (mtf_pthread_func *) mtf_create_prefix, (void *) tcb);

        if (rv != 0) {
           printf("mtf_tcb_start: rv = %d\n", rv);
           perror("mtf_tcb_start");
        }
        assert (rv == 0);

        return 1;
}

int
mtf_tcb_yield(void)
{
        sched_yield();
}


void mtf_tcb_exit(void)
{
        mutex_tcb *current;

        current = mtf_tcb_ptr;

        mtf_lock_release( ((mutex_tcb *) current)->lock );

        #ifdef __osf__
                _longjmp(current->prefix_state, 1);
        #else
```

```
                (mtf_scb_field(tcb_free))(current);
                pthread_exit((void *) NULL);
        #endif
}


void
mtf_exit(int status)
{
        fflush(stdout);
        exit(status);
}


/*
 * block current tcb and schedule a new tcb
 */
int
mutex_block (mutex_container *b_container, mtf_lock *lock)
{
        mutex_tcb *b_tcb;

        /* get current thread */
        b_tcb = mtf_tcb_ptr;

mutex_tcb_put(b_tcb, b_container);

        pthread_cond_wait(&(b_tcb->cv), lock);
}

/*
 * unblock a tcb
 */
int
mutex_unblock (mutex_container *b_container)
{
        mutex_tcb *b_tcb;

        /* get tcb from container */
b_tcb = mutex_tcb_get(b_container);

        pthread_cond_signal(&(b_tcb->cv));
}
```

Appendix C

SemThreads Source Code

C.1 Client Code

C.1.1 semgen.mtf

```
#
# semgen.mtf
#
# Mezcla client description file for SemThreads

# Preamble

set name semgen
set client_inc semclient.h


# Data

data tcb sem_tcb {
        int id;
        mtf_lock lock;
        int exited;
        struct sem_tcb *join_tcb;
        void *join_value;
        struct sem_tcb *next;
}

data tcb_container sem_container {
        struct sem_tcb *head;
        struct sem_tcb *tail;
}

# Code

code sched sem_container {
        set init sem_container_init
        set put  sem_tcb_put
        set get  sem_tcb_get
        set head sem_tcb_head
}
```

```
codel sync sem_container {
        block sem_block {
                sem_tcb_put(b_tcb, b_container);
        }
        unblock sem_unblock {
                b_tcb = sem_tcb_get(b_container);
        }
}

codel sync sem_tcb {
        block join_block {
                b_container->join_tcb = b_tcb;
        }
        unblock join_unblock {
                b_tcb = b_container->join_tcb;
        }
}
```

C.1.2 semclient.h

```
/*
 * semclient.h
 *
 * mtf - The Mezcla Thread Framework
 * Copyright (C) 1999 by Gregory D. Benson
 */

#include <stdio.h>
#include "semgen.h"
#include "semqueues.h"
```

C.1.3 semqueues.h

```
/*
 * semqueues.h
 *
 * mtf - The Mezcla Thread Framework
 * Copyright (C) 1999 by Gregory D. Benson
 */


/* this is needed to allow unoptimized compilation */
#ifndef MTF_INLINE
#define MTF_INLINE extern __inline__
#endif


/* TCB container functions */

MTF_INLINE int
sem_container_init(sem_container *container)
{
        container->head = NULL;
        container->tail = NULL;
}


MTF_INLINE int
sem_tcb_put(sem_tcb *tcb, sem_container *container)
{
```

```
        tcb->next = 0;
        if (container->tail == 0) {
                container->head = tcb;
        } else {
                container->tail->next = tcb;
        }
        container->tail = tcb;
        return 1;
}

MTF_INLINE sem_tcb *
sem_tcb_get(sem_container *container)
{
        sem_tcb *tcb;

        if ((tcb = container->head) != NULL &&
            ((container->head = tcb->next) == 0)) {
                container->tail = 0;
        }

        return tcb;
}

MTF_INLINE sem_tcb *
sem_tcb_head(sem_container *container)
{
        return container->head;
}
```

C.1.4 semthreads.h

```
/*
 * mtf - The Mezcla Thread Framework
 * Copyright (C) 1999 by Gregory D. Benson
 */

/*
 * semthreads.h - a simple thread system based on semaphores
 */

#include <stdio.h>
#include <stdlib.h>
#include "semgen.h"

/* datatypes */

typedef struct sem {
        int value;
        int blocked;
        mtf_lock lock;
        sem_container container;
} sem;


/* functions */

void sem_start(mtf_tcb_func *func, void *arg, int id);
void sem_istart(int id);

void sem_create(sem_tcb **tcb, mtf_tcb_func *func, void *arg, int id);
void sem_join(sem_tcb *tcb, void **join_value);
void sem_exit(void *value);
```

```
int  sem_getid(void);

void sem_init(sem *s, int value);
void sem_destroy(sem *s);
void sem_P(sem *s);
void sem_V(sem *s);
```

C.1.5 semthreads.c

```
/*
 * file: semthreads.c
 * date: 1999-05-02
 *   by: GDB
 * desc: semthreads - a simple thread system based on semaphores
 */


#include <stdio.h>
#include <stdlib.h>
#include "semthreads.h"

#undef  MTF_INLINE
#define MTF_INLINE
#include "semclient.h"

/* functions */

sem_tcb *
sem_tcb_alloc()
{
        return (sem_tcb *) mtf_malloc(sizeof(sem_tcb));
}

void
sem_tcb_free(sem_tcb *tcb)
{
        /* free is done in sem_join */
        /* mtf_free((void *) tcb); */
}

void *
sem_stack_alloc(size_t size)
{
        return (void *) mtf_malloc(size);
}

void
sem_stack_free(void *stack)
{
        mtf_free(stack);
}


void
sem_start(mtf_tcb_func func, void *arg, int id)
{
        sem_tcb *tcb_main;
```

```
        mtf_init(sem_tcb_alloc, sem_tcb_free, sem_stack_alloc, sem_stack_free);

        tcb_main = (sem_tcb *) mtf_malloc(sizeof(sem_tcb));
        tcb_main->mtf_stacksize = mtf_stacksize;
        tcb_main->mtf_stack = NULL;
        tcb_main->next      = NULL;
        mtf_lock_init(tcb_main->lock);
        tcb_main->exited = 0;
        tcb_main->join_tcb = NULL;
        tcb_main->join_value = NULL;
        tcb_main->id = id;

        mtf_start(tcb_main, (mtf_tcb_func *) func, arg);
}


void
sem_istart(int id)
{
        sem_tcb *tcb_main;

        mtf_init(sem_tcb_alloc, sem_tcb_free, sem_stack_alloc, sem_stack_free);

        tcb_main = (sem_tcb *) mtf_malloc(sizeof(sem_tcb));
        tcb_main->mtf_stacksize = mtf_stacksize;
        tcb_main->mtf_stack = NULL;
        tcb_main->next      = NULL;
        mtf_lock_init(tcb_main->lock);
        tcb_main->exited = 0;
        tcb_main->join_tcb = NULL;
        tcb_main->join_value = NULL;
        tcb_main->id = id;

        mtf_istart(tcb_main);
}



void
sem_create(sem_tcb **tcb, mtf_tcb_func *func, void *arg, int id)
{
        sem_tcb *new;

        new = sem_tcb_alloc();
        new->mtf_stacksize = mtf_stacksize;
        new->mtf_stack = NULL;
        mtf_lock_init(new->lock);
        new->next = NULL;
        new->exited = 0;
        new->join_tcb = NULL;
        new->join_value = NULL;
        new->id = id;

        mtf_tcb_init(new, (mtf_tcb_func *) func, arg);

        if (tcb != NULL) {
                *tcb = new;
        }

        mtf_tcb_start(new);
}
```

```
void
sem_join(sem_tcb *tcb, void **join_value)
{
        mtf_lock_acquire(tcb->lock);

        if (tcb->exited == 0) {
                /* printf("sem_join: blocking\n"); */
                join_block(tcb, &(tcb->lock));
        }

        tcb->join_tcb = NULL;

        if (join_value != NULL) {
                *join_value = tcb->join_value;
        }

        mtf_lock_release(tcb->lock);
        mtf_lock_free(tcb->lock);

        mtf_free((void *) tcb);
}


void
sem_exit(void *value)
{
        sem_tcb *p;

        p = mtf_tcb_ptr;

        mtf_lock_acquire(p->lock);

        p->exited = 1;
        p->join_value = value;

        if (p->join_tcb != NULL) {
                /* printf("sem_exit: unblocking joining thread\n"); */
                join_unblock(p);
        }

        mtf_tcb_exit(&(p->lock));
}


int
sem_getid(void)
{
        return mtf_tcb_field(id);
}


void
sem_init(sem *s, int value)
{
        s->value = value;
        s->blocked = 0;
        mtf_lock_init(s->lock);
        sem_container_init(&(s->container));
}


void
sem_destroy(sem *s)
```

```
{
        s->value = 0;
        s->blocked = 0;

        mtf_lock_free(s->lock);
        sem_container_init(&(s->container));
}


void
sem_P(sem *s)
{
        mtf_lock_acquire(s->lock);
        if ((s->value) > 0) {
                (s->value)--;
        } else {
                (s->blocked)++;
                sem_block(&(s->container), &(s->lock));
        }
        mtf_lock_release(s->lock);
}


void
sem_V(sem *s)
{
        mtf_lock_acquire(s->lock);
        if ((s->blocked) > 0) {
                (s->blocked)--;
                sem_unblock(&(s->container));
        } else {
                (s->value)++;
        }
        mtf_lock_release(s->lock);
}
```

Appendix D

Mezcla Code for SR

D.1 Client Code

D.1.1 srgen.mtf

```
#
# srgen.mtf
#
# Mezcla client description file for SR

# Preamble

set name srgen
set client_inc srclient.h


# Data

data tcb proc_st {
    /* struct proc_st { */      /* process descriptor */

    enum pr_type ptype;         /* type of process */
    enum in_type itype;         /* type of invocation if a PROC */
    char *pname;                /* proc name, or "body" or "final" */
    char *locn;                 /* current source location (when blocking) */
    int priority;               /* process priority */
    Sem wait;                   /* sem for initial completion */

    Ptr   sr_stack;             /* process context and stack */
    /* Mutex stack_mutex; */    /* is the stack free yet? */

    /* status and blocked_on are protected by sr_queue_mutex */
    int status;                 /* process status */
    Procq *blocked_on;          /* pointer to list process is blocked on.
                                 * only used by awaken and sr_kill, when the
                                 * JS has a handle on the proc (and thus no
                                 * other JS can), so it does not need to be
                                 * protected.
                                 */

    Bool should_die;            /* is someone trying to kill this? */
```

```
    Sem waiting_killer;         /* if so, is blocked on this */

    Rinst res;                  /* resource process belongs to.  This res has
                                 * a mutex, which is grabbed before we use this
                                 * pointer to change any of the resource's
                                 * fields.
                                 */
    Proc procs;                 /* list of processes for resource.  res->mutex
                                 * is locked when this is manipulated. */
    Cob cob_list;               /* list of co stmt blocks.  Only used with
                                 * cur_proc, so no mutex is needed. */
    Invb next_inv;              /* next pending invocation to examine.  This is
                                 * only used with cur_proc or through a proc
                                 * field in a locked class, so it doesn't need
                                 * to be protected here. */
    Bool else_leg;              /* in in statement with else leg */

    /* The next and next_else lists are not protected, since a proc can only
     * be on one (protected) list at a time and only one job server could
     * have grabbed it from a queue at any time.
     */
    Proc next;                  /*  ready, blocked, or free list */
    Proc next_else;             /*  next process with else option */

    /* mtfsr fields */
    Func func;
    long arg1;
    long arg2;
    long arg3;
    long arg4;


    Rinst cur_res;       /* current resource */


    /* mtf fields */
    unsigned stacksize;
    mtf_lock lock;
}

data tcb_container procq_st {
        Proc head;
        Proc tail;
}

data ecb private_st {
    /* struct private_st { */ /* private variables for each job server */
    int rem_loops; /* remaining loop traversals before resched */
    /* Rinst cur_res; */ /* current resource */
    /* Proc cur_proc; */ /* current process */
    /* Proc old_proc; */ /* previous current process */
    /* char *switch_stack; */ /* for swapping stack locks */

    int my_jobserver_id; /* jobserver ("OS" thread) ID */
    int io_handoff;      /* flag for switching to I/O server */

    /*
     * js_queue_depth tracks the number of grab_queue_mutex calls
     * that have not been matched by the returning give_queue_mutex
     * call.  Thus, if the depth is 0 on entry grab_queue_mutex actually
     * grabs the mutex, and then it increments js_queue_depth.  If
     * this is 1 upon entry give_queue_mutex releases the mutex.  This
     * is necessary because the access to these queues happen at all
```

```
     * levels, and to add parameters telling if we possess a certain
     * queue would be both ugly and error prone.  (Note that being
     * private means that each job server has a copy.)
     */
    int js_queue_depth;
}

data ecbinit b_ecb {
        b_ecb->rem_loops = sr_max_loops;
        b_ecb->my_jobserver_id = b_ecb->mtf_id;
        b_ecb->io_handoff = 0;
b_ecb->js_queue_depth = 0;
}


# Code

code sched procq_st {
        set init sr_queue_init
        set put  sr_enqueue
        set get  sr_dequeue_first
        set head sr_queue_head
}


codel sync procq_st {
        block sr_block {
            /* LOCK_QUEUE ("block"); */
            b_tcb->blocked_on = (b_container);
            b_tcb->status = BLOCKED;   /* after changing blocked_on */
            b_tcb->next = NULL;
            sr_enqueue (b_tcb, b_container);
            /* UNLOCK_QUEUE ("block"); */
            DEBUG (D_BLOCK, "r%06lX *%06lX block   p%06lX",
                  CUR_RES, b_container, CUR_PROC);
        }
        unblock sr_awaken {
            /* LOCK_QUEUE ("awaken"); */
            b_tcb = (b_container)->head;
            (b_container)->head = b_tcb->next;
            if (b_tcb->next == NULL)
                (b_container)->tail = NULL;
            b_tcb->status = READY;
            b_tcb->next = NULL;
            /* UNLOCK_QUEUE ("awaken"); */
            DEBUG5 (D_ACTIVATE, "r%06lX awaken   p%06lX prio %d from %06lX",
            CUR_RES, b_tcb, b_tcb->priority, &(b_container), 0);
        }
}

codel sync procq_st {
        block sr_block_iop {
      /* do nothing */
        }
        unblock sr_awaken_iop {
      /* not used */
        }
}
```

Appendix E

Mezcla Code for Java

E.1 Client Code

E.1.1 jtgen.mtf

```
#
# jtgen.mtf
#
# Mezcla client description file for Kaffe's jthreads

# Preamble

set name jtgen
set client_inc jtclient.h


# Data

data tcb _jthread {
        unsigned char                   status;
        unsigned char                   priority;
        void*                           restorePoint;
        void*                           stackBase;
        void*                           stackEnd;
        jlong                           time;
        struct _jthread*                next;
        struct jmutex*                  mutexlock;


        struct _jthread*                nextQ;
        struct _jthread*                nextlive;
        struct _jthread*                nextalarm;
        struct _jthread**               blockqueue;
        unsigned char                   flags;
        void                            (*func)(void *);
        int                             daemon;

        /* this one is simply thread specific data, or a cookie -
         * used to hold the current Java thread
         */
        void*                           jlThread;
```

```
}

data tcb_container jthread_q {
        struct _jthread *head;
        struct _jthread *tail;
}

data ecb jthread_ecb {
        /* nothing for now */
}

data ecbinit b_ecb {
        /* nothing for now */
}


# Code

code sched jthread_q {
        set init jthread_container_init
        set put  jthread_tcb_put
        set get  jthread_tcb_get
        set head jthread_tcb_head
}

codel sync jthread_q {
        block jthread_block {
                jthread_tcb_put(b_tcb, b_container);
        }
        unblock jthread_unblock {
                b_tcb = jthread_tcb_get(b_container);
        }
}

codel sync jthread_q {
        block jthread_cv_block {
DBG(JTHREAD, dprintf("jthread_cv_block: start\n"); )
                jthread_tcb_put(b_tcb, b_container);
                jmutex_unlock(b_tcb->mutexlock);
        }
        unblock jthread_cv_unblock {
                b_tcb = jthread_tcb_get(b_container);
        }
}
```

Bibliography

[1] The SR programming language, version 2.3.1, January 1999. `http://www.cs.arizona.edu/sr/`.

[2] H. Abelson and G. J. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[3] M. Accetta et al. A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, June 1986.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[5] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, August 1986.

[6] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, California, 1989.

[7] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–70, February 1992.

[8] G. R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, California, 1991.

[9] G. R. Andrews. Personal communication, February 1997.

[10] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings, Redwood City, California, 1993.

[11] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[12] G. R. Andrews, R. D. Schlichting, R. Hayes, and T. D. M. Purdin. The design of the Saguaro distributed operating system. *IEEE Transactions of Software Engineering*, 13(1):104–118, January 1987.

[13] M. S. Atkins and R. A. Olsson. Performance of multi-tasking and synchronization mechanisms in the programming language SR. *Software – Practice and Experience*, 18(9):879–895, September 1988.

[14] D. F. Bacon and A. Lowry. A portable run-time system for the Hermes distributed programming language. In *Proceedings of the Summer 1990 USENIX Conference*, pages 39–49, June 1990.

[15] H. E. Bal. *Programming Distributed Systems.* Silicon Press, Summit, New Jersey, 1990.

[16] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[17] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[18] H. E. Bal and A. S. Tanenbaum. Distributed programming with shared data. *Computer Languages*, 16(2):129–146, 1991.

[19] H. E. Bal, A. S. Tanenbaum, and M. F. Kaashoek. Orca: A language for distributed programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[20] J. Barnes. *Programming in Ada 95.* Addison-Wesley, Wokingham, England, 1996.

[21] P. Barton-Davis, D. McNamee, R. Vaswani, and E. D. Lazowska. Adding scheduler activations to Mach 3.0. In *Proceedings of the USENIX Mach III Symposium*, pages 119–136, April 1993. Also as Tech report UW-CSE-92-08-03, University of Washington, Department of Computer Science and Engineering.

[22] G. D. Benson and R. A. Olsson. *The Design of Microkernel Support for the SR Concurrent Programming Language*, chapter 17, pages 227–240. Languages, Compilers, and Run-Time Systems for Scalable Computers. Kluwer Academic Publishing, Boston, MA, 1996. B. K. Szymanski and B. Sinharoy (editors).

[23] G. D. Benson and R. A. Olsson. Towards microkernel support for the SR concurrent programming language. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1513–1524, Sunnyvale, CA, August 1996.

[24] G. D. Benson and R. A. Olsson. A portable run-time system for the SR concurrent programming language. In *Proceedings of the Workshop on Runtime Systems for Parallel Programming*, pages 21–30, Geneva, Switzerland, April 1997. Held in conjunction with the 11th International Parallel Processing Symposium (IPPS'97).

[25] G. D. Benson, R. A. Olsson, and R. Pandey. On the decomposition of run-time support for concurrent programming languages. In *Proceedings of the Workshop on High-Level Programming Models and Supportive Environments*, pages 86–94, Honolulu, Hawaii, April 1996. held in conjuction with the 10th International Parallel Processing Symposium (IPPS'96).

[26] B. Bershad, E. Lazowska, and H. Levy. Presto: A system for object oriented parallel programming. *Software – Practice and Experience*, 18(8), August 1988.

[27] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, December 1995.

[28] J. Beveridge and R. Wiener. *Multithreading Applications in Win32: The Complete Guide to Threads.* Addison-Wesley, 1996.

[29] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H. Bal, and F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experience with Distributed and Multiprocessor Systems IV*, pages 213–226, San Diego, California, September 1993. USENIX.

[30] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[31] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, November 1986. In *ACM SIGPLAN Notices*, 21(11).

[32] A. P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 181–193, 1985. In *ACM Operating Systems Review* 19(5).

[33] J. Bloomer. *Power Programming with RPC.* O'Reilly & Associates, Inc., Sebastopol, California, 1991.

[34] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. *Programming under Mach.* Addison-Wesley, Reading, Massachusetts, 1993.

[35] E. Cooper and R. Draves. C threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, Department of Computer Science, 1988.

[36] D. E. Culler and J. P. Singh with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc., 1999.

[37] F. Barbou des Places, N. Stephen, and F. Reynolds. Linux on the OSF Mach3 microkernel. Technical report, OSF Research Institute, Grenoble, France and Cambridge, MA, January 1996.

[38] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, May 1965.

[39] E. W. Dijkstra. *Programming Languages*, chapter Cooperating Sequential Processes, pages 43–112. Academic Press, NY, 1968.

[40] E. W. Dijkstra. The structure of the "THE" multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[41] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings*

of the Fifteenth Symposium on Operating Systems Principles, pages 251–266, Copper Mountain, CO, December 1995.

[42] R. A. Finkel, M. L. Scott, Y. Artsy, and H. Chang. Experience with Charlotte: Simplicity and function in a distributed operating system. *IEEE Transactions on Software Engineering*, (15):676–685, 1989.

[43] B. Ford, G. Back, G. Benson, J. Lepreau, O. Shivers, and A. Lin. The Flux OS Toolkit: A substrate for kernel and language research. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, pages 38–52, St. Malo, France, October 1997.

[44] B. Ford and Flux Project Members. *The Flux Operating System Toolkit*. Department of Computer Science, University of Utah, 1996. `http://www.cs.utah.edu/projects/flux/oskit/html`.

[45] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, October 1996. USENIX Association.

[46] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, pages 97–114, January 1994. Also technical report UUCS-93-022, University of Utah, Department of Computer Science.

[47] A. Forin, J. Barrera, M. Young, and R. Rashid. Design, implementation, and performance evaluation of a distributed shared memory server for Mach. In *Proceedings of the Winter USENIX Conference*, pages 229–243, January 1989.

[48] I. Foster and K. M. Chandy. Fortran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 25(1), 1995.

[49] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.

[50] V. W. Freeh. A comparison of implicit and explicit parallel programming. Technical Report TR 93-30b, Department of Computer Science, University of Arizona, 1993.

[51] R. Gebala and C. McNamee. The implementation of the SR concurrent programming language. Technical Report TR95-2, Department of Computer Science, California State University, Sacramento, 1995.

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

[53] R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983.

[54] M. Haines. On designing lightweight threads for substrate software. In *Proceedings of the Annual Technical Conference on UNIX and Advanced Computing Systems*, Anaheim, California, January 1997. USENIX.

[55] M. Haines and K. Langendoen. Platform-independent runtime optimizations using OpenThreads. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 460–466, Geneva, Switzerland, April 1997. IEEE.

[56] H. Härtig, M. Hohmuh, J. Liedtke, S. Schönberg, and J. Wolter. The performance of u-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 66–77, Saint-Malo, France, October 1997.

[57] D. Hildebrand. An architectural overview of QNX. In *Proceedings of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, April 1992.

[58] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*. Academic Press, New York, 1972.

[59] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[60] P. Hudak. Para-functional programming. *Computer*, 15(2):60–71, 1986.

[61] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[62] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[63] Institute for Electrical and Electronic Engineers. *POSIX P1003.1c, Threads Extension for Portable Operating Systems*, 1995.

[64] Institute for Electrical and Electronic Engineers. *POSIX P1003.1c, Standard for Threads Interface to POSIX*, 1996.

[65] Intermetrics, Inc., 733 Concord Ave, Cambridge, Massachusetts 02138. *The Ada 95 Annotated Reference Manual (v6.0)*, January 1995. `ftp://sw-eng.falls-church.va.us/public/AdaIC/standards/95lrm_rat`.

[66] S. C. Johnson. YACC — Yet another compiler-compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.

[67] M. Kaashoek, A. Tanenbaum, S. Hummel, and H. Bal. An efficient reliable broadcast protocol. *ACM Operating Systems Review*, 23(4):5–19, October 1989.

[68] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An interoperable framework for parallel programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[69] J. Kepecs and M. Solomon. SODA: A simplified operating system for distributed applications. *ACM Operating Systems Review*, 19(4):45–56, October 1985. Originally presented at the Third ACM SIGACT/SIGOPS Sypmposium on the Principles of Distributed Computing, August 27-29, 1984.

[70] D. Keppel. Tools and techniques for building fast portable threads package. Technical Report UW-CSE-93-05-06, University of Washington, Department of Computer Science and Engineering, May 1993.

[71] R. E. Kessler and M. Livny. An analysis of distributed shared memory algorithms. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, June 1989.

[72] G. Kiczales. Foil for the Workshop on Open Implementation. Technical report, Xerox PARC, 1994. `http://www.parc.xerox.com/PARC/spl/eca/io/workshop-94/foil/main.html`.

[73] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD UNIX Operating System*. Addison Wesley, 1996.

[74] X. Leroy. *LinuxThreads*, 1996. `http://sunsite.unc.edu/pub/Linux/docs/faqs/Threads-FAQ/html/`.

[75] X. Leroy. *LinuxThreads*, 1997. `http://pauillac.inria.fr/~xleroy/linuxthreads/`.

[76] M. E. Lesk and E. Schmidt. Lex – A lexical analyzer generator. Computer Science Technical Report #39, Bell Laboratories, Murray Hill, NJ, 1975.

[77] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[78] J. Liedtke. Fast thread management and communication without continuations. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 213–221, Seattle, WA, USA, April 1992.

[79] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[80] B. Liskov and R. Scheifler. Gaurdians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming, Languages, and Systems*, 5(3):381–404, July 1983.

[81] D. MacKenzie. *Autoconf*. Free Software Foundation, Inc., 2.8 edition, 1996.

[82] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 110–121, Pacific Grove, CA, October 1991.

[83] Microsoft Corporation and Digital Equipment Corporation. *Component Object Model Specification*, October 1995.

[84] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[85] B. Mukherjee, G. Eisenhauer, and K. Ghosh. A machine independent interface for lightweight threads. Technical Report GIT-CC-93/53, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1991.

[86] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[87] R. A. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 2–11, 1996. Also in ACM SIGOPS Operating Systems Review, Volume 30, Number 5, December 1996, pages 2–11.

[88] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addision-Wesley, Reading, Massachusetts, 1994.

[89] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[90] Larry L. Peterson, Bruce S. Davie, and Andrew C. Bavier. *x-Kernel Tutorial.* 1996. `http://www.cs.arizona.edu/classes/cs525/tutorial/tutorial.html`.

[91] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *Proceedings of the Winter USENIX Conference*, Dallas, TX, 1991.

[92] C. Provenzano. *MIT PThreads*, 1996. `http://www.mit.edu:8001/people/proven/pthreads.html`.

[93] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, Cambridge, MA, 1986.

[94] T. Rühl, H. Bal, G. D. Benson, R. Bhoedjang, and K. G. Langendoen. Experience with a portability layer for implementing parallel programming systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Sunnyvale, CA, August 1996.

[95] E. Schonberg and B. Banner. The GNAT project: A GNU Ada 9X compiler. In *Proceedings of the TriAda Conference*, pages 48–57, December 1994.

[96] K. Schwan, H. Forbes, A. Gheith, B. Mukherjee, and Y. Samiotakis. A Cthread library for multiprocessors. Technical Report GIT-ICS-91/02, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1991.

[97] M. L. Scott. The interface between distributed operating system and high-level programming language. In *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[98] M. L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, 13(1):88–103, January 1987.

[99] M. L. Scott. The Lynx distributed programming language: Motivation, design and experience. *Computer Languages*, 16(3/4):209–233, 1991.

[100] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[101] E. G. Sirer, P. Pardyak, and B. N. Bershad. Strands: An efficient and extensible thread management architecture. Technical Report UW-CSE-97-09-01, University of Washington, Department of Computer Science and Engineering, 1997.

[102] Sriram Srinivasan. *Advanced Perl Programming*, chapter 17 Template-Driven Code Generation. O'Reilly & Associates, Inc., Sebastopol, California, 1997.

[103] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Prentice-Hall, Inc, 1992.

[104] R. E. Strom et al. *Hermes: A Language for Distributed Computing*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[105] R. E. Strom and S. Yemini. NIL: An integrated language and system for distributed programming. *ACM SIGPLAN Notices*, 18(6):73–82, June 1983.

[106] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5), May 1990.

[107] Sun Microsystems. *The Java Development Kit 1.1*, 1999. `http://java.sun.com/products/jdk/1.1/`.

[108] B. K. Szymanski. *Parallel Functional Languages and Compilers*. ACM Press, New York, New York, 1991.

[109] A. S. Tanenbaum, R. van Renesse, H. van Stareren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experience with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[110] The MPI Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputing Applications*, 8(3/4), 1994.

[111] G. Townsend and D. Bakken. *Porting the SR Programming Language*. Department of Computer Science, University of Arizona, 1994. From the SR distribution: `http://www.cs.arizona.edu/sr/`.

[112] U.S. Department of Defense, Washington, D.C. *Reference Manual for the Ada Programming Language*, January 1983. ANSI/MIL-STD-1815A.

[113] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 26–40, December 1991. Also in ACM SIGOPS Operating Systems Review, Volume 25, Number 5, October 1991, pages 26–40.

[114] P. Vishnubhotia. Synchronization and scheduling in ALPS objects. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 256–264, New York, 1988. IEEE.

[115] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, California, second edition, 1996.

[116] T. Wilkinson. Kaffe — a free virtual machine to run Java code. `http://www.kaffe.org/`.

[117] N. Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 3rd corrected edition, 1985.

[118] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.

[119] Y. Yokote and M. Tokoro. Concurrent programming in Concurrent Smalltalk. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming*, pages 129–158. MIT Press, Cambridge, MA, 1987.

[120] S. Zhou, M. Stumm, K. Li, and D Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, September 1992.