A Multi-Threaded Extension to a High Level Interactive Statistical Computing Environment

by

Duncan Walter Temple Lang

B.A. (University College Dublin, Ireland), 1989
M.A. (University College Dublin, Ireland), 1990

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Statistics

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

John A. Rice, Chair
David R. Brillinger
John M. Chambers
Michael A. Harrison

Fall 1997

The dissertation of Duncan Walter Temple Lang is approved:

Chair Date

Date

Date

Date

University of California, Berkeley

Fall 1997

A Multi-Threaded Extension to a High Level Interactive Statistical Computing Environment

Abstract

A Multi-Threaded Extension to a High Level Interactive Statistical Computing Environment

by

Duncan Walter Temple Lang
Doctor of Philosophy in Statistics

University of California, Berkeley

John A. Rice, Chair

A multi-threaded application is one which is capable of performing several tasks concurrently. This work describes the design and implementation of a user-level multi-thread extension to an environment for statistical data analysis based on the interactive, persistent, functional language S. These user-level threads are structured to utilize low level threads, native to the operating system. We outline some of the possible uses of this threaded environment in both single and multiple-processor machines and discuss extensions to the work which will allow the use of a network of machines for distributed computing. The implementations of user and internal functionality based on threads with other approaches are contrasted. We describe the programming interface to both the user level threads and synchronization mechanism. Finally, we outline the design of the internal structures that control the run-time behavior of these threads.

The work here represents low level primitive facilities for an environment such as S to support threads. The ideas can be generalized to other languages such as Matlab, Xlisp-Stat, etc. Also, future work can capitalize on these primitives to further simplify the development of threaded algorithms and applications. These primitives have been implemented in the newest version of the S environment and language created by Lucent Technologies, Bell Labs Innovations.

John A. Rice
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

I am very grateful to the members of my dissertation committee for their time and comments on my work. I would especially like to thank John Rice for supporting and encouraging me to study what might be considered a "non-classical" statistical dissertation topic. Additionally, I would like to thank Phil Spector and Leo Breiman for their involvement in the early stages of this work.

I am indebted to John Chambers and Lucent Technologies for the opportunities they have provided me as a summer intern and consultant over the past 4 years. It is an excellent environment to do research. John is responsible for suggesting the "Graphical User Interfaces in S" project (of which the work on threads is a small part) as a potential dissertation topic. Additionally, he has taught me much and is a wonderful person to work for and with. I am very grateful.

My stay in the statistics department in Berkeley has allowed me to learn many things, not all of them academic in nature. I am grateful for my involvement in the TILE project which allowed me to experience a view of software design and development different from my experience with John Chambers and Lucent Technologies. In particular, I would like to thank Deborah Nolan and Ofer Licht for their tireless patience and persistence in the face of adversity, and Leo Breiman who initiated the project.

The members of the Statistical Computing Facility, in particular Ofer Licht and Rick Kawin, have provided me and others with a wonderful computing environment. Their talents are a significant part of the department and are often under-utilized and under-appreciated. Ofer offered invaluable advice on so many occasions and helped considerably in clarifying my thoughts and providing intelligent ways to achieve certain features. Thanks also go to Sara Wong and the staff of the Statistics Department in Berkeley for taking care of all the administrative material.

On a more personal note, I would like to thank those people who made my life in Berkeley more complete and fulfilling. These people include Melissa Sanchez and her parents. Also, Sean O'Riain has been and is a wonderful friend to me. Similarly, Elinda and Sam Buttrey, Sue Service and Kurt Buck have been very good to me and made life fun. Also, I would like to thank all my fellow students in the department. And finally, thanks to my family.

Chapter 1

Introduction

1.1 Background

Computer users are becoming increasingly familiar with the idea that two or more applications can be in existence and performing tasks concurrently. Individuals can be writing a document in a word processor, reading mail and extracting data from another application without having to terminate one application to move to another. A common paradigm is to be performing lengthy computations *in the background* while doing other more immediate work interactively. A *multi-tasking* environment or operating system is one that supports concurrent applications. In such an environment, two or more processes compete for the machine's resources such as CPU, disk access, etc. which are managed and allocated by the operating system. Most modern operating systems support this facility and allow applications to be simultaneously active rather than insisting that only one is active and the others dormant. The operating system schedules the different processes, making each active for a particular period via a low level *context switch*. This allows computations in one or more applications to proceed while the user reads mail or displays a graph, etc. in another.

This multi-tasking environment is adequate for many users' purposes. Applications have been extended to support multiple views — such as spreadsheets or documents — of which only one is active at a given time. However, it is rare to find applications that can do two or more tasks simultaneously. If one wanted to do two operations within the same application simultaneously, one could create two independent instances of the application and manage the interaction oneself. The absence of the ability to multi-task *within* an application is especially important in the context of an application which is itself a large environment in which one might perform several subtasks as part of an overall project or task. This is the typical case for statisticians in an interactive environment analyzing one or more datasets or experimenting with statistical methods.

Users typically interact with an application in order to accomplish some overall task. This task might be divided into several sub-tasks, some of which must be completed in a specific order and others that can be done at any stage in the overall task. For example, an initial statistical analysis of a dataset might involve:

1. reading the data into the application
2. computing a variety of summary statistics and plots
3. exploring some modeling approaches for the data
4. examining the output and diagnostics of fitted models
5. pursuing other statistical estimators for a subset of the fitted models.

The task is this whole analysis procedure made up of these individual steps, each of which also consists of multiple lower level sub-tasks. The top-level steps are typically performed in the order specified and perhaps a sub-sequence of the steps iterated. However, the order of the lower level sub-tasks within each step may not be important. This is the typical model of interaction for users of interactive statistical languages in which the user can experiment with different procedures in order to understand a dataset or statistical estimator, etc.

While the operating system may support multiple processes, we are unaware of any environments with the capability of performing multiple within-application user level tasks simultaneously. This thesis describes the modifications to the statistical environment and language S necessary to allow multiple low level internal tasks to be performed concurrently by the application, and more importantly to allow the user to dynamically specify high level concurrent tasks using the language itself. There are two types of tasks within each S process: the low level internal tasks that perform bookkeeping operations to maintain the state of the application, and the high level user tasks which are performed by the process. While this work reorganizes the S environment to use threads for its internal computations, the ability for the user to specify high level tasks to be performed concurrently is the critical feature of the work. The redesign of S described in the thesis allows all these tasks — low and high level, internal and user specified — to execute concurrently.

The mechanism by which two or more tasks can be in progress at the same time is a *thread*. A thread is the abstract notion of an independent stream of execution of program instructions within a process. Each thread has its own call stack. Local variables within a procedure behave as they would in a single-threaded application — they are local to that procedure within the thread. Each thread is a structure in the environment capable of processing instructions. The instructions are specified usually in the form of a routine which may include conditional branches and all of the usual flow control. The thread proceeds when active and ignores the presence of other threads until it must communicate with any of them. Effectively, a thread is to a process as a process is to an operating system. An application that has more than one thread is said to be multi-threaded.

The operating system[1] schedules these threads as it does processes. A thread becomes active via a context switch and continues from where it was last executing.

Rather than using a single process with multiple threads, one can use multiple processes each with a single thread to perform the same tasks. The most significant difference between these two approaches is that multiple threads within a process share the same memory and name space within the application whereas independent processes do not. This sharing of memory is what makes threads both useful and complex. A multi-threaded application is instantiated as a process that starts with a single thread which creates other threads. These other threads may have different starting points and instructions which allow different tasks to be performed. However, they can communicate efficiently and transparently using the shared memory. The separate process approach must use more structured inter-process communication (IPC) mechanisms to communicate, which can make some operations quite complicated and awkward. Additionally, the operating system must activate each of the processes with a context switch, giving rise to considerably more overhead than the corresponding action for threads within a process.

Threads can be used on both single- and multi-processor hardware without modifications to the basic task structure, be it high or low level. The scheduling algorithm determines which thread(s) should be made active and associates each with an available processor. This allows the threads to proceed concurrently and makes processors available to idle threads when others cannot proceed while they await the availability of resources. In the presence of more than one processor, two or more tasks can operate in parallel and significant performance improvements can be realized.

The purpose of this work is to extend the functionality of the S environment and language to support threads and thread synchronization. We aim to do this without modifying (significantly) the semantics of the existing language. Much of the work in this document was motivated from the POSIX threads (Pthreads) application programming interface (API) as described in [25] and [29]. This standard, adopted by the IEEE, allows threaded applications to be ported without modification to other machines offering this interface. An alternative "standard" is the UNIX/Solaris interface which is a subset of the POSIX standard. This interface and the differences between it and the Pthreads interface are described in [30]. This work utilizes not only the ideas from these APIs but also implementations of either of them to provide the low level thread facilities which provide the concurrency and scheduling of the higher level threads we create. Use of these types of thread implementations allows us to abstract the S level threads to an implementation independent level.

This work originated from another project which aimed to provide tools for graphical user interface (GUI) development within S [38]. As we describe below, threads lend a useful structure

[1]This is not the case for non-kernel threads which are scheduled by user level libraries within a process and not by the operating system. However, for the purpose of this discussion, the important point is that the threads are scheduled by some external entity.

to a GUI application. That project also provides proof of concept for this thread extension and of the general utility of threads. In some senses, GUIs and threads are opposite ends of the language spectrum in terms of the audience by which they are used. Threads are most useful for advanced users due to the potential complexity of synchronization, whereas GUIs are primarily used by those who wish to avoid details of a language and the necessary alternative to the command line input style. Threads are also useful for task separation in client-server applications. Before offering some examples, we outline the primary advantages and disadvantages of threads.

1.2 Advantages of Threads

Multi-threading offers several benefits.

- Efficiency through parallelization of a task. A single task that can be divided into largely independent sub-tasks can be executed by harnessing the potential of multiple processors. Since each thread can be executing simultaneously on a dedicated CPU, the overall time for a task can be greatly diminished depending on the number of available processors. The improvement in performance can be close to dividing the total length of time for the task by the number of available processors and is only reduced from this optimal rate by the overhead in managing the threads. Many statistical computations are of this form with potential for significant performance gains via parallelization.

- Event handling in GUIs and client-server scenarios. Event driven applications are now very common in the form of graphical user interfaces (GUIs) and servers in a client-server setup. These are characterized by the presence of an event queue on which messages are passed to the application and a suitable action or handler dispatched. Typically, the next event is processed when the handler has returned control to the event loop. If the handler does not return for a significant period of time, the application becomes unresponsive to the other events for the duration of that handler. Threads allow us to execute the handler in a separate thread allowing the next event to be processed almost immediately. In the server, a thread can be created for each client ensuring that a lengthy request from one client does not block the progress of all the other clients.

- Code Development Simplification. Some applications, including S, must monitor streams for input, responding when input is present. Also, they must perform some action at regular intervals such as flashing a message, animation or garbage collection. These tasks coexist with the primary task for which the user has invoked the application. Typically, handling this diverse set of tasks involves building knowledge of each into the code in some manner. For example, commands to process the event loop can be inserted at numerous points in the routine associated with the main task to ensure that these other tasks are processed. Threads allow each task to be coded independently of the context in which it will be used. Each task is executed in its own thread and operates as if it were the only task in the process. Thus no knowledge of these ancillary tasks need be embedded into the other tasks. This modularity greatly simplifies the structure of the code and facilitates initial development. Additionally, it allows new tasks to be easily added as an extra independent task without restructuring or modifying the existing code. In a related manner, primary tasks can make the intermediate results available for other threads to process and display as the primary tasks proceeds. These other threads can allow a user to monitor the state of the computations in the other thread and potentially even control it dynamically. The primary worker thread need not be aware of any other threads performing these ancillary actions but must merely broadcast the intermediate results.

- Input and Output Improvement. Even on a single processor machine, multi-threading an application can yield performance improvements, in spite of the small overhead of managing the threads. This happens when the application must perform some activity that communicates with the rest of the system, such as disk access, and that system resource is not immediately available. As the thread performing this task waits for access to the resource, other threads can proceed avoiding this otherwise idle period.

- Multiple Processes and Multiple Threads. Using several processes to perform multiple tasks involves the operating system managing these processes and having to perform context switches to make each process active at the appropriate time. Context switches for multiple threads within a single process incur significantly lower overhead. This fact explains the term *Light Weight Process (LWP)* often used to describe a thread. This overhead can be further reduced depending on the implementation of the low level thread facility which can be external to the operating system's kernel.

1.3 Disadvantages of Threads

In spite of the potentially large benefits, statisticians and application developers have been relatively reluctant to utilize the multi-threaded paradigm. There are two basic reasons for this. The first is that the hardware required to yield the performance improvements is relatively expensive and accordingly not commonly available. Secondly, other benefits such as those mentioned above are not considered adequate relative to the complexity of developing multi-threaded code. This is perhaps the most important reason for the lack of popularity that threads have suffered.

The advantages of multiple-threads are that they share the same memory within a process and databases within the S environment. However, this is also the source of the complexity.

The generally accepted doctrine that the use of global variables is undesirable takes on an increased significance in a multi-threaded application. Since two or more threads have access to the same variable or memory address, both can modify it concurrently, giving rise to different (and frequently erroneous) outcomes which depend on the scheduling of the threads. Such situations are termed *race conditions* and must be avoided by formal procedures in the code which provide *synchronization* between the threads. Since this involves thinking about the running application in a very different way than traditional single-threaded applications, casual developers often feel the effort is not warranted. Instead, other approaches are used such as multiple processes that communicate in some manner such as pipes, etc. Synchronization requires a certain element of experience to understand the nature of race conditions. Once identified, however, the tools to remove these race conditions are relatively simple and become easy to use when one is familiar with the nature of asynchronous execution. Validating a program's correctness is difficult in the threaded context due to the potential for undiscovered race conditions.

The complexity is enhanced by the lack of tools that are available to assist in the development process. Debuggers and profilers, such as **gdb** and **gprof**, do not typically provide support for threads. While much research has been done on automatic parallelization or threading by compilers, these have only become commonly available in the past few years. Additionally, for high-level interpreted languages the relevance of this research must be investigated further.

1.4 Examples

In this section we briefly outline examples of both numerical and application programming and how they relate to threads. The numerical examples should give the reader a flavor of how computations might be arranged for a multi-threaded system which would improve the performance in the presence of multiple processors. The application examples should illustrate how easy it becomes to add features to a larger statistical environment without having to modify the source code. These examples are simpler to program in themselves using threads and also can be transparently added to a running process without complex consideration of the other tasks in that environment.

1.4.1 Numerical Computations

Cross-Validation and The Bootstrap

Two commonly employed general statistical procedures are cross-validation and the bootstrap. Both employ iterated re-sampling in order to estimate parameters of interest in a non-formula based manner. Cross validation involves creating k disjoint subsets of the data by randomly arranging the n observations into k groups. For each group, the remainder of the data (less the observations in this group) is used to re-fit the model and estimate the parameters. Then the prediction error for the observations in this group are computed. The prediction errors are then aggregated across groups to compute the mean squared prediction error (MSPE). For each of the k groups, the estimation of the model parameters for the restricted data set can be done in a separate thread. The results are merged when all the threads complete.

The bootstrap is a similar procedure that can be used in many circumstances to estimate more general parameters than the MSPE. The idea is simple and explained in more detail in chapter 4. The basic procedure invokes creating a sample of size n from the original data $X_1, \ldots, X_n$ by drawing with replacement. Then the relevant statistic(s), $T(x)$, are computed for this sample. This is repeated B times so that we have B instances of the statistic $T(x)$. This allows us to estimate the empirical distribution of this statistic and estimate parameters from it. The implicit assumption is that sampling from the empirical distribution or original data and constructing the distribution of the statistic is analogous to sampling from the population and doing the same.

Again, each of the iterations or a fixed number of the iterations can be performed in a separate thread. When these threads have completed, we have B values from which we can compute statistics of interest. Since this procedure can be applied to estimating statistics $T(X)$ that might involve lengthy computations (such as optimizing a likelihood function, fitting a liner model, or even recursively bootstrapping), the performance gain from using multiple processors could be significant.

Iterative Computations

In general, any procedure that involves vectorization or loops can be parallelized into multiple subtasks. Two relatively straightforward examples of this type of computation are the EM algorithm [15] and optimization.

Optimization algorithms are commonly required in a statistical computation to maximize a likelihood function in order to estimate parameters of a stochastic model. The standard optimization algorithms use an iterative approach in which each iteration is characterized by the following simple steps

1. Start at a point θ_i.
2. Compute information (from the first and second derivatives of the function) at θ_i
3. Calculate θ_{i+1} by updating θ_i according to some formula based on the derivatives in 2.
4. start iteration $i + 1$ starting from 1.

Rather than maximize the likelihood function, we typically minimize the negative of the log of the likelihood function since this usually simplifies the form of the functional to a simple sum of numerous terms. In many cases, both the form of this function and first and second derivatives are also sums of components which can be expensive to evaluate for a given value of θ_i. Threads allow these independent computations to be performed in parallel and allow the resulting values to be aggregated when each thread has completed.

The EM algorithm allows missing data to be imputed using a parametric approach. The algorithm uses the actual observed data to compute the expected value of the missing data using a context specific model. This is the 'E' step. This new data is then used in the 'M' step to maximize the likelihood to compute new parameter estimates for this model. These parameter estimates are then used in the the next iteration of the 'E' step and the procedure repeated until convergence of the missing values is achieved. In most situations, the computation of the 'E' step involves computing k sufficient statistics. These can be done in different threads. Similarly, the computations in the likelihood maximization can be threaded as above, or simply the computation of the components in the summand parallelized.

Iterative computations in which the output from the i^{th} form the inputs to the $i+1^{\text{th}}$ step cannot be parallelized in this way due to the sequential nature of the iterations. However, frequently each iteration can be parallelized, such as in the computation of the elements in the Hessian in an optimization problem.

1.4.2 Application Examples

The following examples are not usually directly associated with the role of a data analyst. However, now and in the future, statisticians must offer their expertise in all aspects of data analysis which include mechanisms for collecting data, providing easy to use tools for estimation and presentation, etc. [11]. These are not typically numeric computations but environment or application type tools. Again, threads can make the development of these easier and allow them to be added to an existing system without the need for redesign. We discuss some examples of these tools in the next few paragraphs.

Data Acquisition from a Device

While statisticians are frequently presented with previously gathered data, many situations in which statistical analyses are employed involve the dynamic acquisition and real-time analysis of data. Frequently, devices are setup to intermittently measure values and pass them to a computer which may store them, filter them through another procedure, calibrate the machine based on these values, etc. The following are examples of devices and contexts in which the data may be analyzed statistically as it is received.

1. images from a satellite that are "registered" with previous images to monitor changes in the scene,

2. sound from a microphone used for speech recognition,

3. process quality control devices that "test" the components considered output of the process,

4. wind speed used to update meteorological predictions.

While these examples are diverse, they share the basic characteristic that data are read when they are available from the device and processed in some manner. The steps are shown in figure 1.1. The `reader` class in S [12] is designed to handle situations of this form. A reader object is created to listen on a connection/file for any new input. When such input is received, an S expression or function is invoked to read and process the data. In the threaded environment, each such reader executes in its own thread. Each reader is unaware of the other readers and the S environment need not be aware of the task the thread is performing. In this sense, the structure of the reader is completely separated from the environment and highly modular.

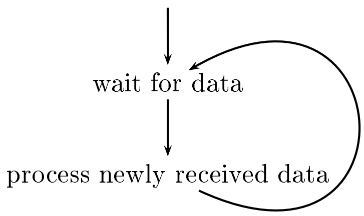

Figure 1.1: "Reader" Procedure

Graphical User Interfaces

As we mentioned when describing the advantages of a threaded environment, GUIs are natural candidates for employing threads. Aside from removing the non-responsiveness of a GUI while a callback is being invoked — by having separate threads for the event loop and event handler — threads can also lead to a simpler structure for GUI development and offer a more intuitive interface and functionality. For example, as an area of a window becomes visible, its contents must be refreshed. As in the implementation of the Java Abstract Window Toolkit [14], a thread other than the event loop or standard callback thread can be used to perform this update. In that way, the remainder of the application need only consider the contents of the window rather than additionally concerning itself with the rendering of these contents.

Applications viewed through a graphical user interface (GUI) often appear as if they are performing two tasks simultaneously. However, usually the task in the active window is the only task being operated on. As each window becomes active, the task is restarted. Threads allow several tasks to be performed within the application concurrently and so users can be in control of multiple tasks with a suitably designed interface.

Remote Databases and Commands

The increasing popularity of the World Wide Web (WWW) has, and more than likely will continue to, lead to the use of distributed data. Retrieving such data can be time consuming. Caching the data locally for the duration of their use is a common approach to minimizing this retrieval time. Threads allow this retrieval to be done asynchronously so that other tasks can be performed more immediately that do not require this data. Threads also allow us to establish data servers within an environment which retrieve and supply these objects to multiple concurrent tasks in an efficient manner. This can simply involve a coordination of the requests by the different tasks to ensure that only a single version of an object is retrieved. Alternatively, a more advanced approach allows the server to preempt the need for an object and to download/create it asynchronously during an idle period in the application. For example, idle periods include the user reading documentation or a splash screen that introduces an application. This concept does not only refer to objects which must be retrieved from a remote source but also objects that can be created locally but are not immediately required. For example, different windows in a graphical interface are likely candidates to be created asynchronously while the user is considering other matters and effectively the application is idle. Creating these windows asynchronously improves response times as the objects will have been created before they are requested by the user. In this way, asynchronous object creation can improve performance in a variety of different fields, even in a single processor environment. Again, threads make this possible and simple.

1.5 History & Current Approaches

Threads have been available in different programming languages dating back to the early 1960s. They were introduced at this point in the language Simula. A program written in this language could consist

> "of a series of interacting processes operating in quasi- parallel as ALGOL stacks within the main program execution." [24]

Only within the last 5 years has the topic of threads become more visible in the form of available books and documents on the Web. One event that led to this was the adoption of the POSIX standard for a thread application programming interface[2] in 1990. This standard allowed different implementations to become available which developers could use to successfully port code to different hardware and operating systems with relative ease. This greatly enhanced the utility of threads, removing the proprietary interfaces to enable portable threaded applications to be developed.

Another significant event in making threads more accessible is the recent success of the language Java [18]. Java offers a relatively high level interface to threads. Due to the popularity of

[2] P1003.1c or ISO/IEC 9945-1:1990c

Java for creating "applets", more people have become familiar with threads without the complexity required by larger and more complicated applications. Unfortunately, Java will probably not be used in the immediate future for performing interactive statistical computations in the same way as S or other environments. This is mainly due to a current lack of availability of pre-programmed, stable statistical procedures in the form of libraries rather than due to a deficiency of the language. This situation should change in the future but must be subject to the extensive testing that existing systems have undergone to assure accuracy.

There are a variety of languages that offer support for threads. These include Sather, Split C, Multi-Lisp and Multi TCL. Split C is not a high level languages and does not offer immediate support for statistical computations. While Sather (or more correctly, the threaded extension pSather) is relatively high level, it also does not provide direct support for statistical programming. Multi-Lisp perhaps could be integrated into the Xlisp-Stat [39] distribution to provide threads in a statistical environment. Multi-TCL is a version of TCL [33] offering the standard Pthread synchronization mechanisms and a few others such as semaphores [19] and reader-writer locks [25]. Since this is based on TCL which uses strings to represent numeric variables, this is not likely to be a viable language for developing statistical applications. In general, these and other languages suffer from the difficulty that they are not high level languages that offer built in support for statistical procedures or data structures commonly in data analysis.

In order to be able to have any success in integrating previously developed statistical libraries into low level languages, thread-safe version of standard numeric libraries such as Linpack, LAPACK, NAG, etc. are required. The NAG libraries support both SMP and distributed style multiple processor support. Within the past two or three years, such multi-threaded libraries have become more common and accessible to the less specialized user.

We are currently unaware of any interactive statistical application that supports user level threads. One of the primary motivations for this work is the belief that data analysts and researchers are averse to using multiple programming languages (for both good and bad reasons) and will attempt to make one or two environments fit most of their computational models. Thus, new features — in particular, threads — have to be incorporated into existing languages.

Much of the work on concurrent computation in the statistical domain makes use of distributed computing. Computer science and other disciplines have differentiated between what are termed *distributed* and *parallel* computation. Parallel computing typically refers to performing multiple tasks within a single process using the shared memory model and threads. Distributed computing refers to performing instructions in different processes, typically on different machines. The cluster of connected machines is probably the most common environment for statistical researchers and one which has the potential to significantly improve performance if the machines can be suitably harnessed. The focus of this work to date is on threads that will work in a single- and multi-processor system and no support has yet been developed for distributed computing.

However, we are confident that relatively minor extensions to the design that we describe in this work will allow the user to transparently evaluate tasks in a distributed environment in the same way as on a single or multi-processor system.

S is commonly used in distributed computing due to its object-persistence approach. Multiple S processes are created, each with its own script which provides the sub-task to it. Communication is via a shared database and the S objects contained within that database. Features in version 4 of S allow a master process to communicate with the slave processes using S connections and to provide the task dynamically. One difficulty with this kind of approach is that there is no synchronization between the different S processes using the commonly shared databases. Thus, we have race conditions which can lead to frustrating results. One commonly encountered problem is that the random numbers generated within each process are identical. This is due to the common value of the seed for the random number generator stored as an object in the shared database. Similarly, race conditions exist if the results from the different processes are stored as individual elements within an object. Instead, they must gathered as separate objects and amalgamated when all the tasks have completed. Also, the S code that implements the different tasks must make liberal calls to `synchronize()` in order to ensure that the contents of the database on disk and the views held by the S processes are consistent. In general, this approach is not very successful as there is no support within the S language for such distributed computations.

A similar facility has been made available through the Matlab environment [40]. This uses multiple processes rather than threads and so incurs the overhead of process communication. One of the interesting features of this project is that it is implemented entirely as an extension developed within the Matlab language itself without modification to the internal source. Again, this suffers from the same difficulties as the naive S approach in that there is only limited synchronization. The project is not complete but it is functional and worth monitoring.

Several lower level software packages assist in developing an application that is to be used in distributed computing. The Parallel Virtual Machine (PVM) package is a portable library which provides significant support for developing distributed applications. As described by its authors,

> PVM enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. PVM transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architecture [21].

The package Linda [9] offers similar support as PVM but uses a different model involving a "tuple space". It allows the virtual sharing of memory without insisting that the underlying hardware physically share the memory. Messages or data are passed between processes through an intermediate location which is the tuple space. This acts as a kind of non-directed message queue similar to the database concept in S.

An even more fundamental or low level package for use in distributed computing is the Message Passing Interface (MPI)[22]. This defines and implements a set of routines that constitute the primitives necessary for message passing. This is not intended to be used directly but as a layer in the distributed environment with additional support required to perform process management and other related matters.

The distributed computing approach frequently suffers from the excessive overhead of passing information between the different processes. The performance gain from using multiple processors is negated due to the necessity for each to wait for data to be communicated between the processes. Increased bandwidth on local networks may reduce this overhead, but naive algorithms are unlikely to yield the performance gains some might expect. Thus, the full utility of the distributed approach may be limited to specialists.

The packages mentioned above are sufficiently low level that no support for statistical computations or data structures is provided. Instead, developers must integrate these facilities into existing software or vice-versa. This is a non-trivial task that must typically be undertaken as a large project rather than as a solution to an individual problem. Few individuals or organizations have the resources to dedicate to such a project.

The Network of Workstations (NOW) concept is a distributed environment that permits high performance distributed computing by offering an integrated operating system across connected machines. This is centered on the idea of a "single view of operating system services across the cluster" [1] The Condor project [17] is another style of distributed computing in which idle machines are exploited to perform operations on behalf of another machine. Both of these allow a user to consider a collection of machines as being controlled by one operating system. While we do expect to provide support for distributed computing using the same user level commands described in remainder of this work, we are unlikely to be able to completely hide the distributed nature of the environment.

Ousterhout delivered a paper at the USENIX conference in 1996 with the evocative title *Why Threads are a Bad Idea (for most purposes)*[34]. The basic idea in the paper is that the complexity of synchronization required by most applications using threads make them an inappropriate tool for development. He suggests that we use events as these are simpler. While the basic argument may be correct in some circumstances, it does not apply to the numerical computation domain. In this context, multiple processors must be used to achieve speedup and events are not an alternative. Similarly for environments like S, the internal structure can be developed in terms of events, but the complexity remains. Instead, this work (especially chapter 5) illustrates that a careful design that minimizes the need for synchronization using events, messages and threads can be successful.

We hope that future work will also allow us to develop a "compiler" or semantic analyzer within S which will allow us to automate the parallelization of certain routines. Perhaps the optimal approach is to have a language such as S be able to translate high level S code into C which uses parallel facilities and offers the functionality of S in a self contained library. This would then allow others to use S in the context of lower level languages such as those mentioned above.

1.6 Overview of Thesis

The work we present in following chapters introduces interactive user level threads in the language S. These have the unusual characteristic that a given thread can be used for multiple sequential tasks and the inter-task state of thread is persistent. This proves to be useful in simplifying the implementation of concurrent statistical algorithms. We develop synchronization mechanisms by which the user can control access by two or more threads to different objects in the environment. This involves a user-level locking mechanism added to the database mechanism in S. The reader should remember throughout the discussion that the material here relates to thread primitives on which a simpler interface can be constructed.

Chapter 2 introduces the basic evaluation model of the S language and the structure of the threaded S environment. The user functions that create and control S threads are then described. Since threads have access to the same objects within the S environment and operate concurrently, unexpected outcomes can materialize due to expressions in different threads being evaluated in an inappropriate order. Avoiding such *race conditions* is the intent of thread synchronization and is discussed in chapter 3. Chapter 4 uses the functionality developed in chapters 2 and 3 to develop in more detail several examples that we introduced earlier in this chapter. Chapter 5 provides a description of the technical issues and decisions made in implementing the threaded environment. This chapter deals almost exclusively with C level structures and algorithms necessary to provide the features in chapters 2 and 3. Finally, chapter 6 describes the current state of the work and future directions we will possibly investigate. The appendix contains a hard-copy of the on-line help associated with each of the S user level functions described in the text.

This thesis does not focus on the performance of the implementation or offer any information about potential performance improvements or gains. The work is considered to be more a design of how threads can be added to a language such as S and is an initial implementation of that design. Performance evaluations and optimization will be undertaken in the future. Also, while our attempt is to make threads available in a high level interpreted language, our work does not make the use of threads transparent to the user. Instead, this is the first and necessary stage in a more ambitious project which includes the ability to offer transparent or automated usage of threads based on the ability to semantically analyze a collection of tasks. This is briefly discussed in chapter 4.

Chapter 2

The S Environment and User Level Thread Application Programming Interface

This chapter starts by briefly reviewing the important aspects of the non-threaded version of the language S and its evaluation model. Next, we offer a high level description of an S user level thread and how it operates and discuss the threaded paradigm in the S environment. This highlights the necessary changes to the evaluation model. We follow this with the main focus of the chapter which is the description of the primitive user level functions and classes which allow the S programmer to use threads. Each of these functions is described with examples of where they might be used. Further examples and practical guidance are contained in chapter 4 and will be further developed in a more tutorial oriented description of the facilities which will be written as a technical report in the near future.

2.1 A Brief Introduction to S Concepts

The work described in this thesis has been implemented as an extension to the newest S language and environment - version 4[1]. Accordingly, a brief introduction to S seems appropriate. We provide only the basic concepts and readers unfamiliar with S should refer to [4]. The ideas are generalizable to several other statistical languages such as Matlab [5] and Xlisp-Stat [39] since these are both functional, interpreted languages that have similar styles of evaluation.

[1]This is in the final stage of beta production by Lucent Technologies, Bell Labs Innovations and is to form the basis of MathSoft Incorporated's Splus 5.0 product in June of next year. Since the code is licensed exclusively to MathSoft, it is not available to the general public and so neither is the major part of this work.

S is an interpreted, functional, object-oriented, dynamic, persistent language that specializes in creation and analysis of data in both a numerical and graphical manner. S is interpreted in the sense that no compilation is necessary to execute an expression. Such expressions contain references to functions and objects containing data and these are resolved at the time of evaluation. One of the main features of the newest version of S, version 4, is that *everything is an object* of some class. This makes it fully object oriented supporting multiple inheritance and most of the standard features in such languages. It organizes class structures and methods differently than other languages, centralizing the methods in a global function name space structure rather than in the object. An interface to C/C++ and Fortran provides access to compiled code allowing for previously developed algorithms to be integrated as full status routines and offering improved performance for critical computations. New additions allow direct manipulation of the S objects in such dynamically loaded C level code. This allows users to extend the base source code of the S environment which is also developed using C.

Perhaps the aspects of that differentiate S from many other commonly used languages are its persistence, name space and evaluation of function arguments. Persistence involves writing objects to disk so that they are available across sessions of the application. S uses the concept of a chapter to collect related data, functions, classes and documents — all valid S objects. Attaching a chapter (or database) makes the objects contained therein visible to the S session. Also, objects can be stored in different chapters by assigning the objects to individual chapters in the *search path*. The search path is a list of chapters attached to the session that allow many different groups of objects to be simultaneously available to the S session. The chapters are used sequentially to find objects referred to in computations and can be arranged in any order at any time during the session. S evaluates expressions or tasks in the traditional manner by creating temporary frames in a call stack. Each frame in the function call hierarchy contains the objects evaluated as arguments to the associated function call and the objects created locally within the function call. Any object not found in this frame or the top-level frame is searched for along the search path. When an object assigned to the appropriate name is found in a database the search is terminated and this object used. Thus, the order of the search path is important in resolving name-object relationships. Not all objects that persist across top-level tasks need be persistent across sessions. S uses the *session database* to store such objects which are discarded at the end of the session. When searching for objects, this database is queried before any of the search path elements. This name resolution mechanism differs from most languages in which objects are either local or global.

S employs lazy evaluation in an effort to be efficient — arguments passed to a function are not evaluated at the time of the function call but only when the argument object is referenced in the computation. This can make the technical internal details of the S evaluator quite complicated but is hidden from the user. Additionally, S allows functions to be defined with default arguments and for functions to determine which arguments were supplied in a particular call and to behave accordingly. Also, a function can be created to take an arbitrary number of arguments using the "`...`" notation. Excluding this "`...`" mechanism, partial argument name matching is used which allows users to identify arguments by name or unique abbreviation and to specify them in any order.

2.2 The Thread Model & Changes to the S Evaluation and Session Model

While the standard version of S does support the concept of a task within a task (a sub-task), it does so in a limited manner. The thread extensions to the S environment and language developed in this work remove these limitations by helping to formalize the concept of an S task and evaluator. The additional functionality offered by threads has marginally changed the nature of the S evaluator which we describe in this section. We start with an outline of how an S thread behaves. Then we indicate how threads interact in the session. Finally, we describe some changes to the S evaluation model due to the presence of threads. The description in this section is a suggestion of how the user should think about the S environment in the context of user level threads.

An S level thread is created using a corresponding C level thread and S evaluator. Additionally, the C level thread maintains a task queue on which a list of tasks to be evaluated by the thread are stored. The thread also maintains its own task history list and uses a thread-specific database which persists across tasks. We describe each of these thread components in the paragraphs below.

When the S thread is created, a corresponding C level thread is also created which waits for an S task to be inserted into its *task queue.* The term task refers to a top-level S expression which is evaluated in a thread. A task queue is an ordered list of tasks. Until a task is inserted into a threads task queue, the S thread is idle. These tasks can be *sent* to this thread by any other thread in the S session in the form of a message or request. The tasks are stored and processed sequentially. When a request containing the task is received by the C thread onto its task queue, the S evaluator proceeds to evaluate it. When it completes this evaluation, it processes the next task in the queue or becomes idle waiting for the queue to be non-empty. In this way, the C thread is in an infinite loop waiting for and extracting tasks from its queue and passing them to the associated S evaluator. When a thread sends a task to another thread it usually does so to have it evaluated asynchronously or "in the background". However, it can elect to wait until the task has been evaluated before processing any other tasks itself.

This task queue approach is very different from the C level thread model [25]. In the latter, a thread is associated with a single task and terminates when it has completed that task. One benefit of the queue approach is that the overhead of creating a new S thread and evaluator for another task can be avoided. More importantly, however, it allows the user to manage the state of a thread across tasks in interesting ways, allowing interesting paradigms to be used which simplify the development and use of threaded applications. The approach also makes an explicit connection between a task and a specific thread and vice versa which is unnecessary in C since there is a single task within a thread.

The evaluation of each task within an S thread is almost identical to that of the non-threaded S. As might be expected, the most significant change relates to the search path and where the thread finds and assigns objects. We discuss the details of this in the next section.

We note here that a task might be consist of an infinite loop. This is true of readers that monitor a connection for input and *timers* which evaluates a given expression at specific intervals. In this case, since the task is a loop, it will never complete and any other tasks in the queue will never be processed. Thus, one has to be careful about which tasks are sent to which threads. We will return to such tasks in section 2.7

2.2.1 Search Path & Name Space

As we mentioned earlier, a significant difference between S and other languages is the name-object resolution mechanism. This uses a list of S chapters as the search path in which objects are found by name and allows each user to modify the search path and, effectively, the behavior of S. S searches for an object by looking in databases in the following order[2]:

1. current evaluation frame in the call stack — frame n in figure 2.1,
2. frame 1 of the call stack — the expression frame [4] in which the task is evaluated,
3. frame 0 (the session frame)
4. each of the databases in the search path

The threaded version introduces changes to the last 2 of these. We start first with the search path.

In the non-threaded version of S, an expression within a task can modify the search path via the `attach()` and `detach()` functions. Any changes persist through subsequent tasks. Accordingly, each thread must have its own copy of the search path to allow modifications to be local to a thread and its tasks. Such changes do not affect the search path of other threads in existence. A newly created thread inherits the search path of its parent thread (almost always the the thread that creates it) and so any changes affect only threads subsequently created within this thread.

Since threads share resources within the S environment/process, it should be possible to establish non-persistent databases which are shared by different subsets of threads. Such a

[2] Other internal databases are used to cache commonly accessed objects but are not listed here since their existence does not alter the semantics.

database is created via the expression

```
d = attach(purpose="data").
```

This can either be attached as the first element in the search path or objects can be explicitly assigned into it via the call

```
assign("name", object, where=d).
```

This stores the object given by `object` under the name `name` in the database specified by the object `d` entered onto the search path above.

In the non-threaded environment, a non-persistent database is automatically created and used to store objects across tasks and inserted as a virtual element of the search path at position 0. This is the session database and can be referenced either using either the `where` or `frame` arguments with the value 0 in the functions `get()`, `assign()` and `exists()`. Similarly, in the threaded environment, each thread has its own non-persistent database. Objects can persist across tasks and yet be local to a thread by using the `frame` argument in these functions with the value 0. The session database is still present in this environment but now must be accessed using the `where` argument and the value 0. (The `frame` argument refers only to the thread's own non-persistent database and can be accessed only by that thread.) This is used to share objects across threads (and hence across tasks) within a session. For example, graphics devices that could be used by all threads might be kept in this database whereas devices used by a single thread would be assigned to the thread's frame 0. This is an aspect of the threaded extension that is not backward compatible. However, the number of functions and scripts that suffer from this change should be small.

Figure 2.1 illustrates the overall existence of databases and evaluation frames in the environment with 3 threads. It also displays the order in which the databases are searched when an object is referenced in a call in frame n by one of these threads. The current evaluation frame is searched first and then frame 1. Next, the thread's own database — frame 0 — is searched. Following this, the usual search mechanism used in the non-threaded S environment takes effect and the session database (referenced by the value 0 for the `where` argument) and the elements in the search path are searched sequentially. If the object is found at any point, the search is terminated and that object returned. The only distinction between the threaded and older, non-threaded environments is the addition of frame 0 in the search path.

2.2.2 S Interface and Task Histories

Typically, a user starts an S session and is greeted with the prompt `>` at which s/he can type S expressions. Additions available in version 4 of S allow the session to be controlled remotely without such an interactive prompt. Similarly, the GUI project that motivated this work

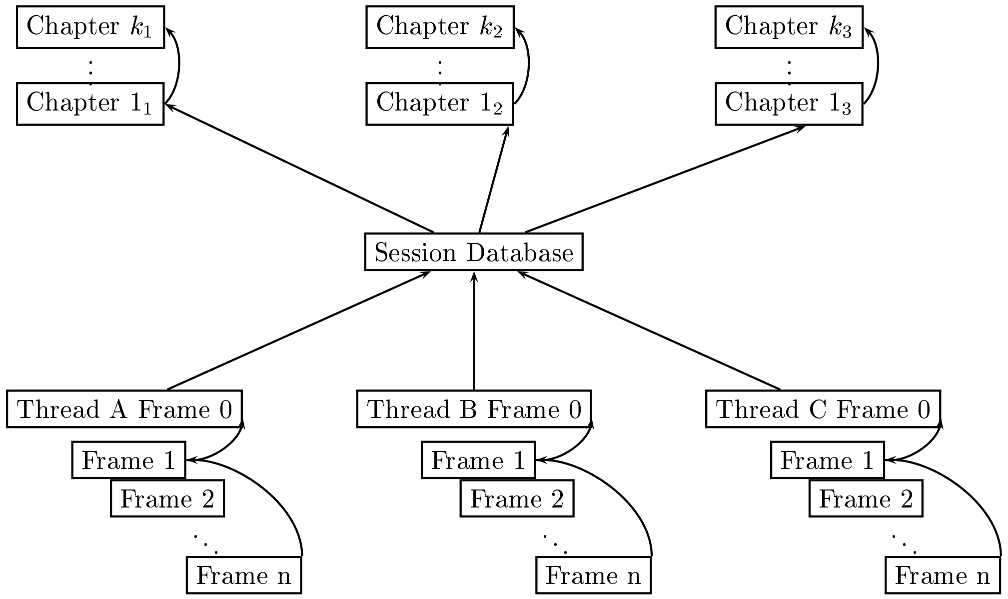

Figure 2.1: Database and Frame 0. Search goes from South to North.

allows multiple prompts to be available in different window components. With these changes and the addition of threads, there is no longer the notion of a single interactive evaluator in the S session. Instead, the S interface can be easily customized to have a variety of very different appearances. The implications of this in the threaded environment include the fact the user can be communicating with one evaluator through several input connections or with several evaluator through one input connection. This impacts the use of the history mechanism and the contents of the journal or logging system implemented via the **.Audit** file.

The S function `history()` returns a list of previously evaluated tasks and optionally allows the user select a task to be re-evaluated or edited. In the threaded context, evaluating a call to this function is ambiguous as the user may wish to consider tasks within the environment as a whole, tasks specified interactively to different threads or simply the tasks evaluated within this thread. Our implementation of this uses the last of these. A list of tasks (and their results) are stored within each thread. The other two uses of the history mechanism can still be supported using functions that search the **.Audit** file. This contains information about each task evaluated and object assigned to a permanent database. This information is cumulated across multiple S sessions and can be used to trace a user's steps. The threaded version of S must now insert the thread identifier into each entry in this log so as to be able to accurately reproduce the user's actions. The tools that currently use this information must be adjusted to take account of the existence of these threads.

2.2.3 Multiple Threads & The Threaded Environment

The following is a brief introduction to many of the concepts we will describe in greater detail in this and subsequent chapter. Above we indicated that one thread can send a task to another. This is the primary mechanism for inter-thread communication. However, the real story involves the *evaluator manager*. This is a separate thread that controls the overall environment. It brokers all requests for communication by one thread with another and also reports the state of the environment used in monitoring and debugging. No thread can communicate directly with the internal contents of another. Instead, all such communication goes through the evaluator manager. We avail of the opportunity provided by this indirect communication to implement a form of access restriction to a thread — the evaluator manager first determines whether the thread from which the request came is entitled to communicate with the destination or target thread.

Daemon threads offer services to all the applications and individual threads such as access to the printer, a graphics device or a centralized help facility. The threaded environment allows multiple S applications to be running within a single S process. Any application can consist of several sub-threads giving rise to groups of threads whose elements are related in some manner.

The evaluator manager is implemented as a C level thread and so consumes resources. However, its tasks are relatively short and so its impact is small. The S evaluator threads have access to almost all the available resources and are scheduled by the low-level thread implementation. If an evaluator is idle, it consumes no resources. S level threads can have priorities indicating relative importance which are used in determining the scheduling of the threads.

Quitting

The S session is terminated with a call to the function `q()`. In the non-threaded environment, being able to call this function assumes access to the unique evaluator and so termination is presumably preceded by suitable cleanup (sub-)tasks. In the non-threaded environment, this is not the case. Other threads perhaps will be in progress when such a call to `q()` is evaluated. Just as in most operating systems [41], all the threads must be notified of this request to terminate and given the opportunity to exit gracefully. This allows these threads to potentially commit assignments and modify objects that might persist across sessions. Accordingly, the function `q()` in the threaded context is a *request* to terminate the session rather than a direct route to termination. When all of these other threads have completed, the process can terminate safely. The evaluator manager is in charge of implementing this shutdown. The function `exit()` is an unsafe way to bring the entire process to completion since it performs the wrapup only for the task in which the call is evaluated.

One other modification to the S engine introduced by threads relates to initialization. Earlier versions of S implemented the initialization directly in C routines. We have now moved this to be a task evaluated by the evaluator manager. This reads an S script from a file which is expected to create suitable threads that will persist after this task. This allows a variety of different startup scripts to be interchanged easily and each of these can implement a different interface. This setup is now also shared by the standard non-threaded S environment.

From this point on, references to S are assumed relate to the threaded version unless otherwise stated.

2.3 S Level Thread Programming

In this section, we describe the primitives available to the programmer to create and manage threads. These primitives allow higher level classes and methods to be constructed which make the programming interface simpler.

2.3.1 Creating and Destroying threads

One creates an S thread using the constructor function `thread()`. This establishes a new evaluator and schedulable C level thread internally and returns an object of class `thread`. This object contains opaque data not to be used directly by the user. This data identifies the thread to the internal C routines that control the S environment and currently is a unique integer associated with the thread. The C level thread is provided by the external library which implements the relevant API and is immediately controlled by the scheduler provided by that library.

The thread is created with an empty task queue. It is idle until a task is sent to it. The `thread()` function takes an argument `expression` which is supplied as an S expression that is considered a single task. If this is supplied in the call, the object is inserted in the newly created thread's task queue and becomes the initial task[3]. In this case, the object returned from the function call is of class `threadTask` rather than `thread`. Objects of this class identify both the thread and the expression which define the instance of the task. Additionally, an integer index related to the thread's task queue is contained in the object which allows efficient matching of tasks rather than comparison of complicated S expressions. The ability to return arbitrary S objects from an S function is one of the strengths of S and might be considered confusing from a user familiar with languages offering strong type checking such as C. In the context of the function `thread()`, we are effectively overloading the function with a different signature and return type. The simple call with no task creates the thread, while the presence of an expression additionally causes the task to be evaluated. These are two different operations combined into one simple interface.

A task passed to `thread()` should be an object of class `call` or `expression`. These are legal S expressions that are created as S objects rather than evaluated at the prompt. There is the usual conceptual difficulty of creating an expression without evaluating it, similar to shell expansion, etc. For example, the call

```
thread(sum(1:10000))
```

will cause the expression `sum(1:10000)` to be evaluated before being passed to the newly created thread rather than having the latter evaluate the expression. This is not what is desired. The two functions `Quote()` and `substitute()` are used in such circumstances in version 4 of S. These allow objects of class `call` to be created for future use. The appropriate version of the expression above would be

```
thread(Quote(sum(1:10000)))
```

The following two examples also illustrate how a thread might be created using these the two expression creation functions `Quote()` and `substitute()`.

```
   # replace data with the contents of the auto data frame
tA = thread(substitute(lm(y ~ .,data=data), list(data=auto)))
   # create an S expression which computes its input objects at
```

[3]It is possible, but unlikely, that another thread could send a task to this newly created thread by "guessing" its identifier based on knowledge of the existing state of the environment and the scheme used to identify new threads. This would rely on uncontrollable timing issues and should not be used as scheduling mechanism.

```
    # evaluation time
  tB = thread(Quote(while(T) { l = readLine(con, n=1);
                               filter(l);
                               update();
                             }))
```

The function `substitute()` replaces S objects referred to in the expression with those supplied in a list of named objects. Quote simply creates an object of class `call` using the unevaluated expression, much like the Lisp `quote` function or ' operator. Calls created with `substitute()` are used to make the data local to the thread whereas those created with the simpler `Quote()` merely create the expression to be evaluated at a future point with the name resolution to be determined at the point in the evaluation. More details on these functions can be found in [12].

The first example above uses `substitute()` to replace the object `data` in the call to `lm()` with the contents of the object `auto`. This is done efficiently with reference counts in the internals of S so we are not creating multiple copies of the `auto` object. The second example simply creates the S object that is the `call` to be used in the `thread`. The objects referenced in the expression (the functions and object `con`) are found when the thread evaluates the expression. Depending on the other arguments to `thread()`, these objects need not yet exist.

Since each task can be a function or an expression of the S class "{"[4] [4] containing any number of sub-expressions, initialization of objects, etc. can be performed as part of the task. This is relatively easy to manage in a language such as S since functions and expressions can be dynamically created or modified in-line without re-compilation. However, it might be more convenient to initialize a thread, populating its local data space with objects before the thread is created (see 3.1.1 below). The argument `data` is used for this purpose and is provided to the `thread()` constructor function. The argument should be an object of class `named` [12]. This is a list of objects each associated with a name. Each of the objects in this list is assigned to the thread's *frame* 0 using the associated name from the list before the first task is evaluated. For example, the result of the two calls below produce identical results in different ways and with different side effects.

```
  tC = thread(Quote(for(i in 1:n) print(i)), data=list(n=10))
  tD = thread(substitute(for(i in 1:n)  print(i), list(n=10)))
```

In the first call, the object `n` is assigned the value 10 in the thread's frame 0 before the task is evaluated. When the task is evaluated and the object `n` resolved, it is found in this thread's database. Other tasks evaluated within this thread may also resolve the object `n` from this database depending on the context of the evaluation. (See 2.1 above for objects are resolved.) The second example uses `substitute()` to replace any reference to the object `n` in the expression and uses

[4] S objects of this class are used to represent collections of expressions.

the value specified in the list supplied as the second argument to `substitute()`. This avoids the side effect of assigning the object to the thread's frame 0. These examples illustrate that the `data` argument provides an alternative mechanism to using `substitute()` to create the task expression and allows objects to be used across tasks locally within a thread.

We have seen that each task is somehow received by a thread into its task queue and that as the current task is completed, the thread processes the next task in that queue. By default, the thread evaluates each task in the same way as parsed expressions are evaluated at the prompt. Namely, the C routine equivalent to the S function `eval()` is invoked with the task's expression as its argument. There are circumstances in which it is more useful to process the task using a function other than `eval()`. For example, a thread might choose to use a different evaluator model so that it resolves object names differently or logs all input from the task queue to a particular connection in a manner not accessible by the function `sink()`[5] (e.g. displays it to a web browser by formatting the output using HTML tags). Another example might have client threads send character strings representing object names to a server thread. The server thread would then interpret these in a special way such as displaying the help for that object.

While each of these can be done given the current S and thread functions, to do so is awkward and clumsy. Each task must be created to perform the relevant action using calls to `substitute()`. Thus each thread that creates the expression to be sent must know what objects to provide in this call to `substitute()`. As an alternative, we provide a more direct mechanism via the `action` argument which allows the creator to override the default use of the standard evaluator. The caller of `thread()` can supply an S function or the name of an S function as the value of this argument. This function is then used to produce the effect equivalent to the expression

```
eval(substitute(action(task)), list(action=action))
```

for each task. The idea is that there are operations that are common to each task sent to this thread and rather than have these operations embedded by the user into each expression inserted onto the task queue, they can be specified once in the action that defines how the thread behaves.

In the same spirit of providing "syntactic sugar" in order to reduce the user's responsibility in creating more complicated thread task expressions, we provide the argument `toplevel`. This allows the user to specify a database which is to be used as the working database (the first element of the search path) for this thread rather than inheriting the parent thread's search path directly. Instead, if this argument is specified, the database is `attach()`'ed to the first position of the new thread's search path.

The argument `data` allows the user to specify a named list of objects which are to be written to the working database before the the thread evaluates its first user specified task. Each

[5]The function `sink()` redirects input or output to a different connection. It works on a stack-based system in which redirections are cumulated so that when the current connection is removed using the call `sink`, the previous connection is used.

element in the list is assigned to the working database using the corresponding name in the list.

Occasionally it is convenient to create a thread without having it start its task immediately. For example, we might create several threads that will be used later in a computation but which must wait for a particular object, such as a remote dataset, to be created. We will introduce general ways to do this in the next chapter using synchronization classes and methods. However, it can be achieved relatively simply in this context by creating each thread with no task specified and then sending that thread the task when one wants the thread to start. Alternatively, it is sometimes more convenient to establish the thread with the task (e.g. inside a loop on which the inputs of the task depends) but have it wait to start processing the task. This can be achieved by specifying the value `F` as the value of the `start` argument. This creates the thread in a suspended more and so while a task is available, the evaluator is frozen. The thread can be continue or resumed using the function `threadStatus()` described below.

Each thread is created by evaluating a suitable call involving the function `thread()` in an existing *parent* thread [6]. The parent for this newly created thread is either the thread in which this call is evaluated or the thread identified by the `parent` argument. This is either omitted or an object of class `thread` created itself from a previous call to `thread()`. The relevance of the `parent` argument relates to how the new thread's attributes are constructed.

Thread Attributes

Certain parameters or objects are used to control the behavior of a running thread. We term these *thread attributes* and group them into a single S object. The attributes are a mixture of:

- S level attributes that govern the behavior of the evaluator,
- parameters controlling the S level thread environment,
- characteristics of the C level thread and its behavior [25].

Currently these attributes include the following objects:

1. connections for input, output and warnings and errors
2. access permissions
3. name for the thread used in place of the thread identifier in output.
4. task history length
5. scheduling priority and control

[6] Except for the first thread which initializes or "bootstraps" the environment and is created directly in the C code.

More may be added later as different implementations of the underlying C level thread mechanism are employed[7].

Before we describe each of these attributes and their relevance, we explain how attributes are calculated for a new thread. Firstly, the attributes from the parent thread are copied. Secondly, the relevant fields in the `attributes` argument of the function `thread()` are copied into the existing attributes. Finally, the values specified in the relevant individual arguments to `thread()` override the corresponding previous settings. This provides the user with flexibility to inherit certain characteristics from the parent thread and to override others without having to first query the relevant fields and carefully construct an object with the relevant values. Thus, specifying a different value for the `parent` argument in `thread()` can be useful if that thread has more appropriate attributes or state than the natural parent thread in which the constructor call is being evaluated.

Connections

Similar to a process, each thread has three standard connections which it uses by default to communicate with the environment: (1) data are supplied to a thread when requested using the input connection. This usually corresponds to standard input for the environment or a batch file or remote control "FIFO". In future versions of S that provide different interfaces for the user, this connection might be a text-field widget, a socket, etc. (2) Output from each task is displayed in an appropriate manner on the output connection and warning. (3) Error messages are written to the error connection. The function `sink()` can be used within a task to redirect input or output to a different connection with the same stack-like properties available in the non-threaded version of S. Such redirection persists across tasks unless until a task alters it.

A new thread typically inherits values for these connection objects unless explicitly specified in the call to `thread()`. Since these and other sibling threads might be running concurrently, care must be taken to synchronize access to these shared connections. Otherwise, output may be interleaved for different threads and input directed to the wrong thread. Coordination between threads is necessary to display any error that occurs in a non-interactive thread and recover appropriately. The high level mechanism for this is discussed below in terms of the function `sendTask()` and also through the synchronization mechanisms introduced in chapter 3. Low level synchronization in the C code ensures that access to a connection cannot be interleaved between two or more threads for certain operations. However, these operations are not likely to be sufficiently coarse for necessary synchronization at the S level to be effective. An alternative approach to such synchronization used in [38] is to have a GUI component for each thread (such as a text or HTML widget) to display the output and errors and a separate component for input (such as a text-field

[7] Currently we are using a user level POSIX thread implementation. However, kernel level threads and UNIX/Solaris threads allow different control to be exercised dynamically.

or data editor). Additionally, input and output can be prohibited by setting the interactive option for a thread, as in

```
options(interactive=F)
```

This makes the thread behave as a background compute server that communicates with the user through an interactive thread.

The following three attributes relate to the S level thread.

Name

In the current implementation, threads are uniquely identified by an integer-valued index. This is hidden from the user using the `thread` and `threadTask` classes. Instances of these classes are returned from calls to `thread()`. Users are encouraged to assign such objects and use them to identify different threads in subsequent calls to other thread-related functions. However, functions described below allow the user to query the state of the environment and sets of threads. The output from these commands provides little association to the thread based on its index. Instead, the `name` attribute allows the user to associate a character string with a thread to make output more descriptive. The `name` attribute has no other utility and dynamic changes to the name are not reflected in previously created `thread` objects.

Task History Length

Each thread maintains its own history list both for efficiency and simplicity. For each task in this list, the associated return value is also cached. In addition to providing content for the `history()` function, this list is also used in synchronizing threads with the `join()` function described below. Since this list could consumes a lot of memory[8], it may be desirable to limit its length. The `maxHistoryLength` parameter specifies the maximum length of list. When the length of the list exceeds this number, the element corresponding to the oldest task is removed.

It is unlikely that the history length parameter will be altered by the user. It is only relevant when there is a significant number of threads and the history list of tasks utilizes memory resources that limit the process. Because of the significance of this task history list to the function `join()`, it is important that the value of this parameter not be so small that `join()` behaves incorrectly. This depends on the history of tasks for each thread involved in the synchronization and is thus context dependent. Again, it is unlikely that this will be a problem as the default is suitably set to cache a large number of tasks in the list.

[8] Note that the list could and should be an S table database or a list stored in frame 0. This is not yet the case for a variety of reasons. However, suitable size/length constraints could be put on such objects.

Access Permissions

C level threaded applications allow all threads to access all shared data and each thread to communicate with other threads it can identify (via some shared data structures). The concept we have for the threaded S environment is that multiple S applications can be running simultaneously unaware of each other. Threads within an application may need to be able to communicate with each other and access each other's attributes but typically inter-application communication between threads is not desired. Since all threads exist at the same level in the S environment, no distinction is made by default between threads from different applications. In order to restrict access to certain threads by other threads, we introduce the `access` attribute. The access permissions for a given thread are used to determine if a thread can call any of the functions described below which involve setting or querying aspects of another thread. These functions include sending a new task, querying the task history list, determining the search path, querying or setting the status, etc. of a thread, etc.

Each time one thread, `tA`, attempts to communicate with another thread, `tB`, it does so by sending a request to the evaluator manager. This then looks at the access permissions for `tB` checking to see if `tA` is included by calling a C routine (or potentially an S function) with the source and target thread, `tB` and `tA` respectively[9], as the arguments. Access is allowed if the routine returns `T` or alternatively it is denied and typically, the command is terminated with an error or warning. In order to do this in a flexible way, the user can set the `access` object for thread `tB` using any of the following objects. The most ubiquitous way of specifying the access permissions is to simply use a boolean. The logical value `T` indicates that all threads in the S environment have access to this thread and `F` indicate that no thread has access (except the evaluator manager which has "root" privileges to all threads). Alternatively, the user can enumerate all threads that are to be granted access. This is done using a list of `thread` objects or more correctly a `threadGroup` object described below. Any thread not in this list is denied access. Finally, the most general form allows for access permissions to be dynamically evaluated. The user can specify an S function for the `access` argument. The return value of this function must be a logical value[10] indicating whether access is granted or denied. The evaluator manager evaluates this expression in the context of the target thread by temporarily suspending that thread. The expression can therefore use objects in the thread's frame 0 and search path. No side effects should be specified in the expression — namely global assignments. The evaluator manager provide the source and target thread objects as arguments to the function before evaluating it. The function can be specified dynamically such as

```
thread(access=function(src,target) {
                  # grant access if the threads are siblings
```

[9]The order may seem juxtaposed but we are determining whether `tB` grants access to `tB`.

[10]The evaluator manager encloses the expression inside a call to `as.logical()` before evaluating it.

```
        parent(target) == parent(source)
    })
```

This example might be used in a daemon server in which a dedicated thread is created as each client connects to the server. Two such threads can communicate in the example above since the function tests the commonality of the parent thread. This avoids us having to store the list of sibling threads in each of the new threads or in a central location. Also, it allows threads created in the future to access previously created threads.

The function should be relatively simple since the evaluator manager is intended to process each of its messages very rapidly. If this is not the case, the performance of the overall environment will be compromised.

Priorities

A priority is an attribute that controls the C level thread. It is used in the low-level thread implementation in determining the scheduling of the different threads. The scheduling priority indicates the relative importance of a thread and, ceteris paribus, which thread should be chosen to become active at the next context switch. Priorities are specified as numeric values on an increasing scale - high values correspond to high priorities. Readers familiar with the UNIX program **nice** will be familiar with this concept. Values range between 0 and the value of the S object `maxThreadPriority`, which is thread-implementation dependent. The value specified is passed directly to the low level thread mechanism rather than being used by the S environment to manage the threads. A brief summary of the common approaches to thread scheduling are described below, and a more complete description can be found in [25].

Priorities are not absolute and depend on other circumstances in the user's overall environment. As the number of processes increase, each S thread is likely to be allocated less of the fixed resources. This is true for both user level thread library or kernel thread implementation. Additionally, different implementations within these types may use different scheduling algorithms making it difficult if not impossible to replicate a given timing behavior. The current kernel level implementation has all threads from all processes competing equally in system-wide contention. User level libraries with which we developed this extension are by definition not scheduled by the kernel but instead within a process. The kernel switches between processes giving them access to the resources and threads within each process compete for this subset of resources. A commonly employed scheduling algorithm used by these thread implementations stores threads with equal priority in a list indexed by this priority. When scheduling decisions are made, an arbitrary thread is chosen from the list with the highest priority such that an element in this list is schedulable. However, other algorithms and variations of this one are commonly used.

When developing most threaded C level applications, the number of threads and their roles are usually known at compilation time. As a result the priorities express a relationship between the fixed set of threads. In the threaded S environment, multiple applications can be in existence simultaneously each with their own priority relationship. All the threads within the S environment compete for the resources rather than within a sub-application. This means that the effect of thread priorities assignments depend not only on other processes being executed on the machine but also on other applications being run within the S environment.

The value of a thread's priority can be changed dynamically. While it can make sense to modify the priority of a thread across tasks, it is usually not necessary and is rarely a good idea when performed within a task. Generally, one considers the priority for one group threads relative to another and use arbitrary values which simply express the direction of importance, rather than the estimate of the difference of importance.

We take this opportunity to provide a brief description of commonly employed scheduling algorithms for threads used by the low level C libraries that provide the native threads in our environment. A more detailed discussion can be found in [25]. A thread can be in any one of three states that affect its scheduling. These are runnable, blocked and active. Active means that thread is proceeding on its assigned CPU. Runnable implies that the thread is ready to proceed but has not been assigned to a CPU, and blocked indicates that the thread is waiting for a resource, such as a lock, to be released by another thread or process before it can be considered runnable. The scheduler maintains a list of runnable threads. This list can be thought of as a ragged array indexed by thread priorities with each element being a queue of waiting threads of equal priority. When a CPU in the system becomes available — either because the thread becomes blocked or the scheduler decides to preempt that CPU's currently active thread - the scheduler decides which thread will next become active. In general, this thread will be taken from the queue of threads associated with the highest priority element in the list of runnable threads. The details and subtleties of these algorithms, of course, differ between implementations and especially between libraries offering user level threads and those providing lower level kernel threads[11].

In short, priority values must be considered crude indications of relative importance between groups of threads and never used as a synchronization tool.

S `threadAttributes` Class

Rather than specifying the individual attribute arguments in calls to `thread()` when creating several threads, it is more efficient and intuitive to create a `threadAttributes` object is created which contains the relevant values. Then this object can be supplied as the `attributes` argument in a call to `thread()`. This class is used primarily as a simple data structure or list in which to hold a collection of related values. The constructor function `threadAttributes()` shares the same names as the attribute arguments in `thread()`. Values in the resulting object

[11]Scheduling for a distributed environment is a more complicated concept but fortunately is rarely of concern since the different machines perform separate tasks that must only communicate rarely and do not compete for shared system resources

corresponding to arguments that are omitted in the call are not replaced by the counterparts in the calling thread. Instead, these are left as default values from the S prototype for this class. The object also contains information about which of the slots were set and which are default values. When the object is used in calls to `thread()` or `setThreadAttributes()`, the default values are replaced appropriately with the calling thread's values or ignored. Specifically, `thread()` uses the values from the new thread's parent. `setThreadAttributes()` is a function used to set the attributes of a thread dynamically. It behaves similarly to `thread()` in that it allows the user to specify a `threadAttributes` value as the `attributes` and to override the values in this object individually with arguments for each of the slots in this class. Only the attributes specified directly in the call or in the `threadAttributes` object are replaced. The others are left unchanged.

Example

At this point, we can show an example of how the functions and arguments described in the previous pages might be used.

```
 # create a new database
l = attach(purpose="data")

 # create a new threadAttributes object to be used in creating
 # all of the threads. This specifies that the working database
 # for each thread will by default be the object ‘l’.
 # Also, it indicates that the thread should only keep information
 # on the last 2 tasks and that the the only other thread to have access
 # to the new thread is the value returned by self(), which is this
 # thread evaluating these expressions
attrs = threadAttributes(toplevel=l, access = self(), maxHistoryLength=2)

k = 10
th = vector("list",k)
for(i in 1:k) {
   # create the thread and add it to the group.
  group = add(group,
                thread(substitute(foo(total),list(total=i)),
                         attributes=attrs,
                         priority=(maxThreadPriority + i)/2))
}

# change the output to a new connection and increase the priority.
```

```
 # for the 4th element in the thread group.
setThreadAttributes(elements(group)[[4]],
                      connections=list(output=file("log")),
                       priority=maxThreadPriority)
```

In the above, we create a new transient database. Then we create a new `threadAttributes` object to be used in creating each element in a group of threads. This limits the size of the history list maintained by each of created threads to 2 and also establishes the transient database as the working database for each thread. Additionally, it limits access to the return value of the function `self()`. This is the `thread` object of the thread that evaluates these expressions. Equivalently, we could have supplied a function

```
function(src, target) {
  target == parent()
}
```

The use of `self()` causes the evaluation to be done before the threads are created rather than dynamically each time the access permissions are determined. Finally, we dynamically set the priority and output connection for one of the threads.

Destroying a Thread

Because of the task queue approach there is no need to destroy a thread since an idle evaluator consumes no CPU resources. However, the thread does consume memory resources associated with the C stack and internal S representation and also requires management. Accordingly, it is better to terminate a thread when one is sure that it is no longer needed. This can be done with the function `kill()` or directly with the function `threadStatus()` called with the value `"TERMINATE"` specified as the `value` argument. Note that such a thread can be used for totally different tasks either directly by the user or by the evaluator manager. The overhead of creating and initializing a thread can be avoided by recycling these threads rather than destroying them. The evaluator manager is entitled to cache unwanted threads rather than destroying them and reusing them in subsequent calls to `thread()`.

2.4 Intra Thread Control

The `thread()` function allows us to create threads and associate a single task with each. Since each task is evaluated by the standard S evaluator (or some specialized evaluator), any S expression evaluated within that task is legal and will behave in the usual manner. There are a few additional functions and modifications to existing functions that can be used to control the evaluation of the task.

The function `self()` returns an object of class `thread` which identifies the current S thread. This can be used in functions wherever a thread object is needed. Several functions use this as the default argument for the target thread to which operations are applied.

Occasionally, a thread that is currently active will decide to defer execution to another thread, giving up its current access to the CPU. The algorithm that schedules the threads will then identify the next thread to become active (which could be the same thread that has just yielded) and associate that with a CPU. The function that performs this deferral is `yield()`. It takes no arguments and indicates to the scheduler that it should make another thread active if possible. It is analogous to the C routine `pthread_yield()`.

At this point, there may be no apparent need need for the function `yield()`. It turns out to be useful when two or more threads are working concurrently to perform some larger task and we want to help ensure that each thread gets approximately equal resources. A simple example of this involves counting in different threads. We create two versions of a function `foo()`. Each counts from 1 through 100 and prints each value. One of implementation calls `yield()` after each iteration. The other does not. In each experiment, the top-level task spawns two threads each with their own (sub)task which call `foo()`. The code and output for each is given below.

```
foo = function(limit) {
  for(i in 1:limit) {
    cat("(",self(),")", i,"\n")
  }
}

k = 10
thread(Quote(foo(k)))
thread(Quote(foo(k)))
```

```
foo = function(limit) {
  for(i in 1:limit) {
    cat("(",self(),")", i,"\n")
    yield()
  }
}

k = 10
thread(Quote(foo(k)))
thread(Quote(foo(k)))
```

Without `yield()`	With `yield()`
(2) 1	(2) 1
(2) 2	(3) 1
(2) 3	(2) 2
(2) 4	(3) 2
(2) 5	(2) 3
(2) 6	(3) 3
(2) 7	(2) 4
(2) 8	(3) 4
(2) 9	(2) 5
(2) 10	(3) 5
(3) 1	(2) 6
(3) 2	(3) 6
(3) 3	(2) 7
(3) 4	(3) 7
(3) 5	(2) 8
(3) 6	(3) 8
(3) 7	(2) 9
(3) 8	(3) 9
(3) 9	(2) 10
(3) 10	(3) 10

The output[12] from these examples differs based on the state of the S environment and what other threads are in existence and also on other processes running on the machine[13]. However, the difference in the outputs displayed here is typical of the effect of `yield()`. For the thread that invoked `yield()`, the resources are balanced more equitably between the threads and the output from each iteration is interleaved in each thread. Note, also how we can use `self()` to identify each thread.

The function `sleep()` causes the calling thread to pause for a specified period of time. Other threads can become active during this period and the sleeping thread consumes few or no CPU resources. This function has been modified in the threaded version to utilize the C routine `select()`. This allows finer time resolution than previously available. `sleep()` takes the number of seconds for which to pause and an optional second argument specifying additional micro-seconds.

2.5 Inter-thread Thread Control

2.5.1 Manipulating the Task Queue

The following functions are intended to allow a thread to communicate with other *target* threads. Each function can also be used to communicate with the calling thread, `self()`.

[12]This is actual output from S, version 4 running on a Pentium 133 Mhz PC using Provenzano's user level Pthread library [36] and the Linux 2.0.14 kernel.

[13]Additionally, the non-yielding example leads to radically different outcomes depending on the number of counting threads (i.e. > 2) and the total to which they count. Space does not permit us to display more than one of these outcomes.

The task queue is the link between one thread and another and indicates what each thread is to do. Every S task[14] eventually processed by an S evaluator is first appended to the end of the associated thread's task queue by the function `sendTask()`. The task queue works on a first in-first out principle, so the order in which the tasks are added to the queue is important. The body of the task is an S expression of class `call` and is passed as the first argument to `sendTask()`. Again, such expressions are created dynamically as S objects, frequently using `substitute()` and `Quote()`.

The second argument to `sendTask()` is a `thread` object identifying the thread to which the task is to be delivered. In the example immediately below, we create a thread with an empty task queue and then create a task using `Quote()`.

```
tA = thread()
sendTask(Quote(for(i in 1:100) print(i)), thread = tA)
```

This is sent to the newly created thread identified by `tA` and inserted into its task queue from which it is removed by the target thread, `tA`. At this point `tA` can become active and proceed with its evaluation of this task and others delivered to its task queue.

We strongly advocate the approach of assigning the thread object returned from a call to `thread()` for use in later calls. However, in interactive exploratory use common in S during which commands are evaluated interactively and added to the body of a developing function, the assignment is sometimes overlooked. Currently, knowing the handle/index of the thread from the S thread object is sufficient to recreate the thread object. The constructor method for the generic `thread()` can be used with a single numeric argument to create an S `thread` object as if it had been returned from the call `thread()` or coerced from a `threadTask` object to a `thread` object. No actual C- or S-level thread is created, just the S object corresponding to the thread identified by this index. For example, the call

```
tA = thread(3),
```

creates the S object representing the third user level thread created in the session. There is no connection between this and other methods for the generic function `thread()` other than the return type.

Using `sendTask()` as above causes the task to be added to the end of the target thread's task queue and no more. This means that `sendTask()` returns and the task is evaluated at some point by the target thread asynchronously or "in the background". In other words, the thread calling `sendTask()` does not wait for the task to be evaluated and is ready to continue its evaluation or next task. When invoked in this way, `sendTask()` returns an object of class `threadTask` if the task was successfully appended to the target thread's queue. Alternatively, if that thread does not

[14]Except the first or bootstrapping task to initialize the process.

exist or the Evaluator Manager determines that the calling thread does not have the appropriate permissions to write onto the target thread's task queue, `NULL` is returned and a warning or error reported depending on the context. These `threadTask` objects are used in the function `join()` and other task queue related operations.

There are circumstances in which it is convenient or necessary to have a task evaluated in one thread, B say, on behalf of another thread, A, while thread A waits for the result of evaluating that task. This occurs when the state of thread B is such that it is best suited to evaluating the task based on its previously evaluated tasks and contents of frame 0 and its `threadAttributes`. For example, if a GUI attempts to display a help file, it might send the request to a Help daemon thread (see below 4.2.2). This might be appropriate since that daemon thread is responsible for caching help documents and managing windows and so might have the relevant text in memory rather than having to perform expensive I/O. Once the document has been read, it will be cached and subsequent calls avoid rereading it again. While it is not obvious why thread A must wait for the task to be completed in this example, there are circumstances in which it may make sense.

The argument `wait` to `sendTask()` is specified as a logical value (`T` or `F`). `T` indicates that the call to `sendTask()` should not return until the task has been evaluated by the target thread. In this case, the return value is the result of evaluating the task. This is almost equivalent to `.Last.value` from the target thread. `.Last.value` is shared across threads and so may be overwritten by another thread before this task is fully complete. Also, the `action` argument to `thread()` is entitled to evaluate other expressions as part of the task that would update `.Last.value`. So, the object returned from `sendTask()` with `wait=T` is more likely to be the one of interest. As we have seen, the default for the argument `wait` is `F`.

The help daemon example suggests a good 'rule of thumb' for simplifying threaded programming. Different threads can synchronize with each other by communicating using tasks rather than other more complicated structures and techniques that we describe in the next chapter. In this example, only the Help daemon thread performs the I/O for reading a help file, so there is no possibility for two threads to check if the text has been cached and both having determined the object was not previously read, attempt to read the file concurrently. Basically, the tasks are created concurrently but evaluated serially by the thread to which they are sent. We may apparently lose the concurrency of evaluating the tasks, but since the tasks need to be serialized there is no concurrency to lose. The synchronization is only required at the task queue level and this is done internally by the evaluator manager.

2.5.2 Removing a Task

As one can add a task to a queue, one can also remove a task from the queue. Removing a task is one necessary procedure in implementing an Undo facility, for instance in a GUI. A GUI (see 4.2.1 below) might have one thread handling user actions and queuing tasks in a second thread. An undo action would remove or cancel the last task if it had not yet been completed. If it had been completed, other actions must be taken. Also, one thread could queue the same task in several threads with the understanding that as the first of these threads started to evaluate that task, the task would be removed from the task queue of the other threads.

The function `removeTask()` is invoked with an expression that (exactly) matches the corresponding expression object passed to `sendTask()`. The second argument is the `thread` object whose task queue is assumed[15] to contain this task. Alternatively, a single argument of class `threadTask` can be specified which provides the expression, thread and perhaps a task index. This index is used in simple and efficient matching of the expression with elements in the target thread's task queue. The value `T` is returned if the expression is matched by an element in the queue and the task removed, or `F` otherwise. The calling thread must have access to the target thread as determined by the evaluator manager.

There are "interesting" timing issues relating to `removeTask()`. As one thread, A, attempts to remove a task, t_1, which is the first element of thread B's task queue, t_1 may be moved from the task queue to become the current task. `removeTask()` is implemented by having the Evaluator Manager communicate with the target thread (B) on behalf of the source thread A. In this setting, the manager prohibits B from accessing its own task queue while it is examining the elements. It starts by examining the current task and if that matches the supplied task expression, it cancels the current task and returns. Otherwise, the manager loops over the elements of the queue performing the same matching to find the relevant task. When the manager has completed its search, it allows thread B access to its task queue. Other forms of synchronization between threads can be implemented at the user level using the synchronization primitives described below.

Querying the Task Queue

In general, one should not be removing tasks from a thread's task queue without knowing the exact expression. If one does not know what tasks are in a thread's queue, more than likely the design of the computations is flawed. However, for interactive computations and in the case of the undo facility above, it is convenient to be able to examine a thread's task queue elements and its other attributes such as its task history list, last result, etc. The following functions allow one to query these values but not set them. Each takes a single `thread` object as the only argument.

The function `currentTask()` returns a `threadTask` object representing the task currently being evaluated. The function `taskQueue()` returns a list of the expressions in the thread's task queue in the same order they will be evaluated. This allows, for instance, the user to interactively remove tasks from the queue such as sending plots to a printer, etc. The thread can be temporarily suspended (using `threadStatus()` below) as the user identifies and removes the relevant task(s).

The function `valueOf()` returns the object that was the result of the last task evaluated

[15] If the task is not contained in this thread's task queue, an error occurs.

by the specified thread. S users will be familiar with the object `.Last.value` in the working database used to store the value resulting from the last top-level expression evaluated. There is a somewhat subtle difference between the functions `valueOf()` and `lastValue()`, which returns the object `.Last.value` from the specified thread. Since `.Last.value` is committed to the thread's working database, and the working database can be shared between two or more threads, the meaning of `.Last.value` can be ambiguous for a particular thread. This is simply due to the race condition between the threads that share the database and the order in which each writes `.Last.value` to it. `valueOf()` allows the user to retrieve the object returned from the last task without this ambiguity. `lastValue()` avoids the user having to use the function `get()` and query the thread for its working database object in order to retrieve the object `.Last.value`.

Finally, a list containing pairs of tasks and return values for the previously evaluated tasks in a given thread is returned by the function `taskHistory()`. This list is stored in the thread, mainly for use by the function `join()` in chapter 3, and its length is governed by the thread attribute `maxHistoryLength`. object. So the length is the minimum of the number of tasks evaluated in this thread and the value of `maxHistoryLength`. This can be used to provide a full record of the top-level activities or tasks of a thread.

One of the primary purposes of these functions that allow querying tasks in a thread is to permit reasonable debugging and monitoring tools to be easily developed. We imagine a display analogous to the UNIX command **top**. This would be a dynamically updating display of the status of a specified subset of the threads in the S environment. The output for each thread would display the current task and its current usage of resources. This might be implemented simply as follows:

```
top = function(...) {
    threads = list(...)
    threads = threadIds(threads)

    while(T) {
      for(i in threads) {
          # output formats the arguments suitably for display in a
          # GUI or whatever.
        output(self(),currentTask(i), threadResources(i))
      }
    }
}
```

The functions `threadResources()` and `output()` would take care of retrieving the thread's resources and displaying them on the thread's standard output connection.

As with `sendTask()` and `removeTask()`, all of these functions above operate through the evaluator manager which ensures exclusive access to the task queue to avoid any synchronization issues. Again, higher level synchronization can be programmed between threads for which it is necessary, such as in a user-defined task scheduler.

Querying the State of a Thread

A thread can be in any of several different states of execution at any given time. Some of these states are user-specifiable, such as SUSPEND, CANCEL, TERMINATE and others are more descriptive states that the thread finds itself in such as IDLE due to a lack of tasks, BLOCKED waiting for access to a common resource or in a user-supplied C routine. The function `threadStatus()` is used to both query and set the status of a collection of threads. An arbitrary list of threads can be provided to `threadStatus()` via the `...` argument. The operation (setting or retrieving the status) is applied to each of the threads identified in this list. These thread elements are processed using the function `threadIds()` which traverses the elements recursively and returns a single level list of `thread` objects. If the argument `value` is missing, the current status for each of the target threads is returned as a `named` object. The position in this object corresponds to the position in the list and the names are the character description of the status. The values are the numeric codes for these status names, corresponding to the elements in `ThreadStatusValues`.

If a value is supplied for the argument `value`, the status of each of the target threads is set to that value, if possible. The argument `value` can be supplied as the numeric code or the character description found in the globally accessible object `ThreadStatusValues`. These include

"UNKNOWN"	"ALIVE"	"BUSY"	"IDLE"
"FINISHED"	"BLOCKED"	"TERMINATE"	"CANCEL.TASK"
"CANCEL"	"CANCELLED"	"SUSPEND"	"SUSPENDED"
"CONTINUE"	"IN.DOT.C"	"IN.DOT.CALL"	"IN.DOT.FORTRAN"

Only those that suggest an action (`SUSPEND`, `CANCEL`, `TERMINATE`, `CONTINUE`) have any effect. The rest are used to describe the current state of a thread.

This function is not vectorized in the argument `value` at this point, so supplying a vector of values for this argument results in a warning and only the first element is used. The function is vectorized in the `...` argument which allows multiple threads or `threadGroup`s to be specified. Simpler functions are provided for commonly assigned status commands which are interfaces to `threadStatus()`. These include `cancel()`, `suspend()` and `kill()`.

A thread has other attributes that can be of interest and facilitate computations. We have described the tree-like relationship between one thread and those threads created by it. Each thread has a *parent* thread that created it or which was associated with it by the `parent` argument to `thread()`. This thread object can be retrieved using the function `parent()`. If no argument is supplied, the target thread is assumed to be the result of `self()` and the computation is done locally by looking at the evaluator's internal information. Alternatively, a list of target `thread` objects can be supplied and the evaluator manager returns the parent object for each of these. Similar to `parent()`, the function `children()` returns a list of thread objects representing each of the children threads of the specified thread. If no argument is supplied, again `self()` is used as the target and the computations done locally.

A thread's search path is the list of databases in which named objects not local to the top-level task can be found during evaluation. A thread can evaluate a call to the function `search()` to retrieve its own search path object. This vector of databases associated with a remote thread can be retrieved directly using the function `searchPath()` by specifying the target thread as the only argument. This function is also a method for the generic function `search()` invoked from the signature

```
search("thread")
```

The difference between using `searchPath()` and the other functions above rather than sending a suitable task such as

```
sendTask(Quote(search()), thread=tB, wait=T)
```

to the target thread illustrates an important point about thread communication and the purpose of the functions introduced above. These functions cause the evaluator manager to look at the target thread's internal structure and compute the necessary information. The calling thread must compete with other threads for the manager's attention by putting the request in the manager's message queue. Using the `sendTask()` approach to communicate directly with the target thread, the task is appended to the end of that thread's task queue. The calling thread will be blocked until the target thread evaluates this expression. If that queue is lengthy or any of the tasks is time consuming, the calling thread could be delayed for an extended period. The evaluator manager is designed to handle a high volume of short messages with a higher turnover than a thread. So having the evaluator manager perform the actions for these functions reduces the length of time the calling thread is likely to be blocked. However, the context should be used to determine which approach is most appropriate. The `sendTask()` approach is suitable if each of the target threads was designed to process very brief tasks to be used as a message queue similar to the evaluator manager.

2.5.3 Thread Groups

Many of the functions described in this chapter operate on a collection of threads. These can be passed to the functions as a list or individually using the `...` argument where appropriate. The `threadGroup` class is a more specific type of container than a list which is used to store associated `thread` objects. Elements of a `threadGroup` object are either `thread` or `threadTask` objects or other nested `threadGroup` objects. Formalizing the notion of a collection of threads allows methods to be dispatched based on this type not possible via the `...` argument. This simplifies the interface to certain functions.

The constructor `threadGroup()` takes an arbitrary number of `thread` objects passed via the `...` argument. These are added as elements of the group. Methods for the generic functions `add()` and `remove()` are available which return the modified `threadGroup` with the obvious changes. These functions take an arbitrary number of `thread` or `threadGroup` objects and match these against the elements in the specified `threadGroup` object. Note that the return value must be assigned to store the changes to the `threadGroup` object. The function `elements()` returns a list of the entries in the `threadGroup`. This can be used to add the elements of a group to another group rather than nesting the `threadGroup` object itself.

`threadGroup` objects can also be used in determining access permissions for a given thread. As we saw above, a thread can be selective about what other threads are able to communicate with it. The collection of permissible threads can be supplied as a `threadGroup` object. This raises an important distinction between the S `threadGroup` class and the corresponding Java and C structures. S has no concept of a pointer[16] and so we cannot specify a `threadGroup` at one point to indicate its contents in the future at the time it is used. Instead, the object is substituted at the point of evaluation. This is why we allow access permissions to be supplied as expressions or calls in addition to `threadGroup` objects. The expressions allow for the `threadGroup` to be evaluated at the time permissions are checked rather than when specified as a value for the `access` argument to `thread()` or `setThreadAttributes()`. The `threadGroup` can thus be changing dynamically during the life of the thread and is evaluated dynamically at each access request.

The current implementation of the `threadGroup` class is exclusively in S. In other words, the information about the contents of a particular group are not stored in the evaluator manager. This was not an obvious choice as in some ways having the information at that level makes other operations simple and more efficient. For example, it allows elements to be added and removed without assignments. The current implementation allows the possibility that while an operation is being applied to one element in the `threadGroup`, another thread in the group cab be performing another operation. Accordingly, the result of the operation may be out-of-date when the function returns. Basically, operations on a `threadGroup` object are not atomic with respect to its elements and so the elements can proceed with their tasks during the operation. This is a basic problem that while less likely if the data were stored in the evaluator manager would still be present. Other synchronization methods must be employed if such a situation causes problems.

Storing the information directly in C structures also moves away from the spirit of the S language. It does facilitate the syntax of modifying the group by adding or removing objects from it. The S implementation requires that the user assign the newly modified object such as

```
group = add(group, tA, tB)
```

Certain syntactic brevity can be employed which allows for replacement methods to be created which would allow

[16] Except using the `raw` class.

```
add(group, tA,tB)
```

in much the same way as the replacement operator `[<-()` works.

As this extension is used in more contexts, we will discover if the `threadGroup` is a useful one and whether an internal implementation is more appropriate for its common uses. The Java language [32] provides a similar construct via the `ThreadGroup` class which is useful in the context of an environment, such as the Java Virtual Machine [27]. It limits the elements to be `Thread` objects.

2.6 Manager Functions

Now that we have discussed the primitives necessary for controlling aspects of a thread we turn to the global S environment. The evaluator manager either knows all information about the threads in existence or can compute it immediately. We outline some of the functions available that are trivial extensions of the existing internal mechanism used as an interface to the evaluator manager.

A list of all threads known to the evaluator manager is returned by a call to the function `threads()`. The list uses names supplied at creation time to identify the threads.

The function `access()` is used to determine whether the source thread has access to information (task queue, search path, history values) in a list of target threads. If the argument `source` is not supplied, the value of `self()` is assumed. An arbitrary number of target threads are provided via the "`...`" mechanism as the second argument. The return value is a vector of the same length as the number of target threads. The i^{th} element of this vector is a logical value indicating whether the source thread has access to the corresponding target thread.

2.7 Readers and Timers

When we first started implementing user level threads in S, the aim was to map the existing classes into this new framework. Two concepts which were in the original version 4 of S and are directly related to threads are readers and monitors [12].

A reader in S is used to listen for input on a connection corresponding to a file descriptor. When no input is available, the reader consumes few or no resources. When input is discovered on the connection, the reader evaluates a user-specified task which typically reads the available input and processes it in some fashion. Recall the earlier examples relating to reading data from devices (see 1.4.2). The critical aspect of the reader concept is that while no input is waiting the reader *sleeps* and other tasks in the environment can be performed.

A monitor is simply a timer that performs a user-specified task when the timer expires. The monitor is set at a time t_0 and expires at time t_1. At this point, the timer becomes active and evaluates the associated task. This may involve resetting the time to expiration to a sometime in the future.

The common concept between readers and monitors is that they are event driven and they share the same essential characteristic that they do not block evaluation of other tasks while they are "inactive". Much effort was invested in developing version 4 of S by John Chambers to allow connections to be non-blocking wherever possible. An event loop forms the heart of the S engine which we will describe in greater detail in chapter 5. This is similar to the X event loop (but additionally admits subtasks) in that it operates identifying a task in the following order:

1. check for input on a connection

2. check for expiration of a timer

3. check if an evaluation is currently being performed

If no task is found, the event loop returns and waits until input on a connection is detected or a timer expires.

The point of this long description is to illustrate that these two concepts are not special features in the presence of threads. Instead, these can be implemented as S functions and the need for the complicated and error-prone event loop removed. If for no other reason, the addition of threads has been useful in simplifying the structure of some of the basic functionality in S.

The reader is simply a `while` loop that reads from the connection using a blocking version of the read routine. This returns in the case of an error or with the input read from the connection. Then the input is processed and the next iteration started.

```
reader = function(con, f) {
   while(!isEof(con)) {  # terminate on end of file
     l = readLine(con)
     f(con,l)
   }
  return(T)
}
```

The waiting for input in the call to read is done at a low level so that it consumes no resources while it is idle and allows other threads to be scheduled to utilize the available resources.

The monitor involves calling the function `sleep()` followed be executing the user-specified expression/task. The thread in which this monitor is evaluated will block but it will consume no resources between t_0 and t_1 and allow other threads to proceed.

```
monitor = function(t,task, reset=T) {
  sleep(t)
```

```
    eval(task)

     # If we want to have a reset, it is simple using sendTask().
    if(reset) {
       task = substitute(monitor(t,task, reset), sys.frame())
       sendTask(task, self())
    }
   return(task())
  }
```

As a result of these simple implementations, there are as yet no subclasses of the S `thread` class making the design and use significantly simpler.

Chapter 3

Synchronization

In this chapter, we introduce the main difficulty involved in threaded applications — synchronization — and the mechanisms available in our extension used to handle this aspect of multi-threaded development. As in the previous chapter, we introduce primitive classes and methods from which higher level structures can be developed. We provide examples of such high-level structures and examples in which they might be used.

Synchronization is concerned with coordinating and serializing access to shared variables and serializing (sub)tasks across threads. It can be divided into three different categories. The simplest synchronization necessary involves synchronizing two or more tasks to be performed for which the order is important. A more usual and complicated version of synchronization involves protecting data from concurrent access or modification by multiple threads. Finally, we examine methods for synchronizing access to objects based on their values.

The basic concepts in our implementation are borrowed from the Pthreads interface. However, we do introduce an interesting mechanism of associating the objects to which access is being synchronized with the object implementing the synchronization, the `threadLock` class.

3.1 Basic Synchronization

The coarsest level of synchronization between threads occurs when one thread must wait until other threads have completed specific tasks before it can proceed. This arises, for example, when one thread creates several children threads to perform different sub-computations that are ultimately combined to produce the result of the overall task. This is basic approach used in implementing parallel numerical computations such as k-fold cross-validation (see 4.1.1). The main thread would create k children each of which would compute the fit on the subset of data removing the k^{th} group of observations.

The function `join()` provides the low-level mechanism to perform this waiting for tasks in other threads to complete. In the POSIX threads specification, each thread has only one task and then it "disappears". S threads, as we have seen, persist across tasks and behave as *task servers*. Because of this, the concept of waiting on a thread is not meaningful for S threads. Instead, we must wait for a task to complete within a specified thread. Accordingly, `join()` takes objects of class `threadTask` which identify the thread and task. The object returned by `join()`'ing on a `threadTask` is the object resulting from having evaluated the expression in that thread. This object is also retrievable using the `taskHistory()` function. Any number of `threadTask` objects can be passed to the function `join()` via the `...` argument. `join()` returns when all of the specified tasks have been completed by the relevant threads. The return value is a list of the objects from having evaluated the tasks. `join()` behaves similarly to functions introduced in the previous chapter allowing the elements of the argument `...` to be supplied as `threadTask` objects or lists or `threadGroup` objects. The objects are converted into a single-level list of `threadTask` objects.

The main synchronization aspects of the cross-validation example are now relatively simple to program using `join()`. In the example below we present an outline of the code. We note that object `data` is assumed to be visible to all the threads by being in a database on the search path and that it is not modified during the computations. (Synchronization methods for ensuring the latter are introduced below.) Also, the numerical computations are performed in a `CVfit()` which are not shown and largely irrelevant for the purposes of this example.

```
CV =  function(data, num.threads=10, ...) {
    # create a list to hold
  threads = vector("list",num.threads)

  for(i in 1:num.threads) {
    task = substitute(CVfit(data, model, i,...),,list(i=i,...))
    threads[[i]] = thread(task)
  }

  on.exit(threadStatus(threads,"TERMINATE")) # ensure all the children
                                             # threads are killed
     # wait until all the tasks have completed
     # and have the return value as an element in the list
     # components.
  components = join(threads)
  return(components)
}
```

While the implementation details are irrelevant for its use, we note that `join()` waits for the completion of the specified tasks without consuming resources. Rather than constantly checking the status of each task (*polling*), the `join()` function sleeps, waiting to be notified when each of the tasks has been completed.

Our function `join()` is similar to the Pthreads `pthread_join()`. However, for reasons specified above, ours relates to tasks rather than threads. Additionally, our implementation can wait for multiple tasks, unlike its Pthreads counterpart. That allows a given thread to wait on only one thread. Our vectorized approach makes it easier for the programmer to use.

3.1.1 Synchronization

Threading as we have described it can greatly simplify code development. Rather than arranging for tasks to be non-blocking when they are waiting for input somehow (e.g. reading from a connection, timers, etc. see 2.7 below) in order to allow parts of other tasks be performed, the threading mechanism takes care of the implicit scheduling details transparently. However, this simplification is realized only when the tasks are independent in that they do not share common resources/objects. When two or more tasks do share resources, each must arrange to access the resources in a coordinated manner to ensure the consistency of these inputs. This aspect of threaded development — object synchronization — has been a major factor in reducing the accessibility of parallelization and threads to the casual user. As Ousterhout puts it, *"threads are hard"* [34]. The difficulty is initially two-fold. Firstly, one must become familiar with the different ways in which lack of synchronization can affect the behavior of threads. This experience allows the developer to then identify the expressions in the code that require synchronization and the possible ways this can be implemented.

A simple example illustrates the additional complexity introduced by the need to synchronize access to shared objects. Consider the scenario in which we have two threads, A and B. The task in thread A includes the expression[1]

```
x = x + 1
```

Similarly, thread B will evaluate the expression

```
x = x - 1
```

Suppose that `x` has the value 10 before either of these expressions is evaluated. Thread A evaluates its expression of interest by first evaluating the right hand side and then assigning this object using the name `x`. The value of this object is 11 and at this point thread B becomes active and evaluates its expression. It will retrieve the value of `x` as 10 since thread A has not yet assigned its updated object to `x`. Accordingly, thread B will continue by assigning the value 9 to `x` and the final value

[1] S does not have an `++` operator as it is a functional language.

of `x` will depend on which of the two threads commit the assignment last. The outcome could be 10 as intended or either 9 or 11.

This non-deterministic outcome due to access by multiple threads of a shared object is called a *race condition*. The presence of race conditions in an function or script make it invalid and can lead to subtle errors in the form of incorrect computations, etc. What is worse is that the errors may not manifest them due certain timing conditions that occur on a particular machine or environment and can be nearly impossible to reproduce. This is the single most difficult aspect of multi-threaded code development but can be simplified using certain tools and experience.

Critical sections and Locking objects

Race conditions identify the existence of certain blocks of expressions in different tasks in which two or more threads cannot be concurrently active. These expression blocks are termed *critical sections* [25]. A critical section is, roughly speaking, a block of expression which must be executed or evaluated as a single or atomic operation with respect to other *associated* critical sections.

The example above can be implemented differently in S. (For example, the task sent to each thread A and B could substitute the value for `i` or provide it as an argument.) However, many situations require critical sections within a task. In the example above, the two threads access the shared object `x` and these expressions define two associated critical sections. However, any other thread, say C, that is also accessing `x` must also be synchronized with the other threads and contains a critical section. This may occur in more than one expression block in a task and/or in more than two threads. Thus we see that important step in identifying the need for synchronization is in identifying groups of associated critical sections. Such a group is used to provide protected access to one or more objects shared across multiple tasks.

A group of critical sections is intended to ensure that only one element (a critical section) within the group is being evaluated at a given time. Before each thread enters a critical section, it must ensure that no other thread is currently executing in any associated critical section. If this is not the case, this thread must wait until all other threads have exited the relevant critical section. Of course, two or more threads may be waiting to enter associated critical sections. In this case, only one may proceed and the others must wait.

The vital property for a set of associated critical sections is that at most one thread can be executing expressions within its critical section and that thread has *mutually exclusive* access to the active status of the critical section group. Accordingly, the POSIX threads interface [25] defines a structure called a *mutex* [25] for this purpose. This acts as a lock to which there is only one key. As each thread prepares to enter a critical section, it attempts to acquire the lock. If it is already held, that thread cannot acquire the lock and waits. As each thread exits a critical section, it releases the lock so that an other thread might acquire it. The thread scheduler determines which of the waiting threads acquires the newly release lock based on the thread priority

In the following sections, we describe the primitive synchronization mechanisms available to the S programmer. In other sections below (see 3.2.3), we illustrate how these can be used to provide higher level mechanisms that provide common synchronization facilities found in other thread implementations.

3.2 The `threadLock` Class

We implement the S level `mutex` feature with the class `threadLock`. Objects of class `threadLock` have the same characteristic as the C-level `mutex` in terms of providing mutual exclusion for associated critical sections. They also offer additional features that are consistent with persistent multi-scoped languages such as S. One can think of a `threadLock` object as being a simple interface to a unique `mutex` and accordingly use it to ensure serialized access to critical sections and objects. Additionally, a `threadLock` object can be considered a database. This allows a group of threads to share objects in a synchronized manner in a central location only visible to elements of that group. The database is a lightweight memory table rather than the usual permanent disk based database and allows tight coupling of the objects and the lock being used to provide synchronized access.

A `threadLock` object is created via a call to the `threadLock()` constructor function. Typically, one thread creates the `threadLock` and then creates the threads that will use it. While creating the children threads, the parent thread passes the `threadLock` object as part of the task. Since the the `threadLock` object is a database, it can be attached to facilitate references to objects in the task expression and also to avoid aliasing such objects by those with the same name in databases earlier in the search path. This can either be done within each new thread as part of the initialization of the task (e.g. using `substitute()`), or by attaching the database in the parent thread and having each newly created thread inherit the search path.

One acquires the lock by calling the function `getLock()` with the `threadLock` object as the argument. The `threadLock` object is passed as the first argument and the call returns only when the lock is acquired. Similarly, the ownership of the lock is relinquished by calling the function `yieldLock()` with the `threadLock` object as the first argument.

At this point, we have presented the necessary tools to be able to examine an example of how `threadLock` objects might be used. k threads are created which each call the function `foo()`. This function computes an object based on its arguments using a function `bar()`. Then it appends one component of this object to the object `x0` and the other to `y0` which are shared across the k threads. It appends these elements to the two `list` objects while holding the lock to avoid race conditions between the threads.

```
k = length(x)    # number of threads to create.
```

```
    # create a list to hold the newly created threadTask objects.
children = vector("list",k)
    # create the threadLock database
lock = threadLock()

    # initialize the shared objects x0 and y0 in the threadLock
  assign("x0",list(),where=l)
  assign("y0",list(),where=l)

  # create each of the k threads
for(i in 1:k) {
 children[[k]] = thread(substitute(foo(x[[i]],y,l), list(l=l, i=i)))
}
```

The function foo() is implemented as follows.

```
foo = function(x,y,l) {
   ans = bar(x,y)         # computations on x and y
   getLock(l)             # acquire the lock so that only this thread
                          # can operate in this critical section

     x0 = get("x0",where=l)       # retrieve the x0 object
      x0[[length(x0)+1]] = ans$x  # append component to this object
      assign("x0", x0, where=l)    # assign x0 back to database.

             # repeat for y0
     y0 = get("y0",where=l)
      y0[[length(y0)+1]] = ans$y
      assign("y0", y0, where=l)

   yieldLock(l)           # release the lock for other threads to
                          # proceed in this critical section.

   invisible(return(NULL))
}
```

As an aside, we note that we can avoid defining foo() and using locks by implementing the above using join(). The code would be as follows:

```
       l = threadLock()
```

```
children = vector("list",k)
for(i in 1:k) {
  children[[i]] = thread(substitute(bar(x[[i]],y,l), list(i=i,l=l)))
}
values = join(children)
 # now separate this into the x and y components of each element in
 # values
 ..
 ..
```

3.2.1 Recursive Locks

One of the common problems associated with using locks is that a thread will attempt to acquire a lock that it already holds. The usual locking facility will attempt to acquire the lock a second time and block, waiting for it to be released. Since that thread already holds the lock, it will be never be released and the thread will be deadlocked. This happens most frequently in nested function calls that operate on the same shared object.

A recursive lock [25] permits the thread holding the lock, and only that thread, to acquire it repeatedly without blocking. The `threadLock` class and methods implement a recursive lock. Calling `getLock()` within a thread that already holds that lock increments a count which stores the depth of the locking. Each corresponding call to `yieldLock()` decrements the count and the lock is only released for acquisition by other threads when the count is 0.

Occasionally, it is necessary to fully release the lock even if the count is non-zero. This arises commonly when an error is encountered. Each acquisition of the lock can be associated with a call to `on.exit(yieldLock(lock))` which ensures that the count is appropriately decremented even in the case of an error. This is true regardless of the depth of the evaluation in S as all the expressions registered with `on.exit()` in the call stack are executed even in the event of an error. However, removing this command from the list of expressions to be executed on exiting an evaluation frame may be non-trivial. A simpler method of ensuring that the lock is released is to employ the argument `all` of the `yieldLock()` function. If this is specified as `T`, the count is ignored and the lock fully released.

The function `lockOwner()` returns the identity of the thread which holds the lock on the specified `threadLock` object. This can be useful when tracking causes for deadlock or data corruption by incorrect usage of locks.

In some contexts it is also useful to be able to attempt to acquire ownership of the lock but to return immediately without blocking if the lock is held by another thread. This might used in the case where a thread can conditionally perform an alternative sub-task if the lock is held to fully utilize the computer's resources. Calling `getLock()` with the value `T` for the `try` argument implements this non-blocking attempt to acquire the specified lock. Alternatively, the function `tryGetLock()` performs the same action by calling `getLock()` with the appropriate arguments. This is analogous to the C routine `pthread_mutex_trylock()`.

We should note that locking arises less frequently in a quasi-functional language than in other types of languages that use references (or pointers). Recalling the evaluation model, we see that the only shared resources are accessed at the top level expression when specified as arguments to the task or directly referenced within the task given the understanding that the object is considered globally accessible. The latter is not good programming practice, especially in S, and should be avoided. The former suffers from synchronization problems introduced by lazy evaluation. An argument is evaluated only when first encountered within a function. Since the value of the argument may come from a shared database, its evaluation may require synchronizing. There is no simple way to do this. The object should be evaluated immediately within a critical section either explicitly as part of the task or in creating the expression using the `substitute()` function. Also, careful choice of data representation and liberal use of combining objects into a composite object or class structure can simplify the synchronization task.

Conditional Locks

The synchronization tools introduced above are sufficient for serializing access to shared objects and executing code. Consider next how we might synchronize two or more threads based on the *value* of a shared object rather than access to the object itself. For example, suppose we implement a version of the `join()` function that has the parent thread wait for its k children to complete a given task. For the purpose of this example, we arrange for each child to increment the value of the object `num.finished` shared by all threads. The parent thread waits until the value of the object equals k. [2]

```
k = 10
num.finished = 0
lock = threadLock()

for(i in 1:k) {
   thread(substitute({foo(i);
                      getLock(l);
                      num.finished = num.finished + 1;
                      yieldLock(l)
                     },list(i=,l=lock)))
```

[2]Alternatively, the i^{th} child might set the i^{th} element of a vector `finished`. The parent thread waits until the expression `all(finished)` evaluates to `T`. This involves very little synchronization other than that provided by the internal database routines in S.

```
}

# wait until num.finished == k
..
```

Since `num.finished` is accessible by all $k + 1$ threads, each must acquire the lock to access or modify it to avoid race conditions as in the previous section. This is done in the task expression passed to `thread()`. The parent thread must also test the condition `num.finished < k` while holding the lock. However, if the condition is false, it must relinquish the lock to allow the other threads to increment the value as they complete their tasks. The following code allows the parent to proceed only when all the children threads have finished their tasks providing one aspect of the behavior of `join()`.

```
finished = F
do {
 getLock(lock)
 if(num.finished < k)
   yieldLock(lock)
 else
   finished = T

} while(!finished)
```

The parent thread iterates in an loop acquiring the lock and testing the condition. It releases the lock if the condition is false and repeats the iteration. Otherwise, it escapes from the loop and proceeds with its other work. This implementation leaves a very small window of opportunity in which other threads can acquire the lock between the iterations of the loop. Secondly, it consumes a lot of resources by *polling* the value of the object in a *busy loop*. This usage of resources is non-trivial as it might prohibit the children threads from ever becoming active and completing depending on the scheduling algorithm and thread priorities.

The above approach is not a viable solution so instead we seek something that has lower unecessary resource utilization and is easier for the user to program. Rather than polling the value of the object(s) in the condition (i.e. `num.finished`), it is better that the parent thread be notified each time one of children threads completes its task. This could be done using `sendTask()` from each child with the following code:

```
sendTask(Quote({num.finished = num.finished + 1;
                if(num.finished == k) do.remainder()}), parent())
```

This approach allows the parent thread to remain idle until each thread completed. At this point, the parent would receive and process the task above. While this is possible in the above example, it is clumsy and not generalizable to situations in which the parent cannot treat the entire task as two separate tasks and so return to process its task queue. This is because each thread must complete its current task before retrieving the next task from its queue. Thus the parent must terminate its task to process those sent from the children or else they would remain in the queue as the parent waited for these tasks to be completed. While a thread can query its own task queue during a task, this is not the intended purpose of that function.

Additionally, we want to ensure that each time the parent thread tests the condition, it holds the lock, relinquishes it if the condition is false, and holds the lock if the condition is true. Our solution is to use the `threadLock` object as a location in which to store the objects shared between the threads and that are used in determining the status of the condition. In the example above, this is `num.finished`. Again, we use the function `getLock()` and specify an additional argument which is the condition expression to be tested. This function arranges that the internal C routines test the condition and perform the necessary locking and unlocking similar to the example above. The thread sleeps, using no resources, after the condition is evaluated returning false. It is notified each time an assignment or removal of an object is performed in the specified `threadLock` indicating that the value of the condition may have changed. On receipt of such an action, the thread re-evaluates the expression and if it returns `T`, the call to `getLock()` returns with the lock being held by this thread.

Using this mechanism, the example above becomes simpler. The task sent to each of the children threads remains the same. The parent now waits for these tasks to be completed using the expression

```
getLock(l, condition = substitute(num.finished == k, list(k=k))
```

This replaces the `do ... while()` construct above and eliminates its inefficiencies.

The condition expression is evaluated when the database is modified in any way. It is evaluated as if it were evaluated using the the S function `eval()` with the same name space. In other words, it creates a new evaluation frame and the objects in the previous frame are invisible. This explains the need for the call to `substitute()` in creating the expression in the example above.

The internal database routines in the S source code provide the necessary notification of the waiting thread(s) of the database modifications. These routines do this using the `mutex` and `condition_variable` structures available by the low level thread interface. The details of how a thread waits for such notification and another thread notifies it is implementation specific for these thread libraries. In general, there are two basic approaches for having a thread wait. Either the thread implements an efficient busy loop constantly checking the value of a shared variable or it sleeps, awaiting a signal from the scheduler to continue. The former approach is typically used when the expected duration of the sleep is short. In the latter approach, the notifying thread must "raise" a signal or notify the scheduler of the change.

The simplification that this mechanism offers necessitates that the object(s) referenced in the condition expression be in the single `threadLock` object specified in the call to `getLock()`. This is a constraint that usually is not an inconvenience.

In the near future, we will implement a more efficient and advanced version of this approach. This involves identifying the objects referenced in the expression (see 4.3) and having the waiting thread be signalled only when any of those objects are altered — not on modifications to any object in the `threadLock`. This avoids the overhead of evaluating complex conditions for each database action. Instead, only relevant events cause the waiting thread to be signalled. This is a natural extension of the `getLock()` function and the `threadLockEvent` class.

3.2.2 ThreadLock Events

In the preceding section, we claimed that the `condition` argument was an S object of class `call`. However, in fact this is a special case of a class called `threadLockEvent`. The idea behind the use of `getLock()` to conditionally acquire the lock is to allow different types of "events" to specify the condition. The class `threadLockEvent` is an abstract class that can be extended so as to describe different types of events that might be useful to allow a thread to monitor the database or `threadLock` object and acquire the lock when such an event is received. These events are not limited to S expressions but also include actions on the database.

Currently, there are four basic `threadLockEvent` classes. The first of these is `threadLockExpressionEvent`. This is the one used above which is specified by an S expression which is to be evaluated when any modifications to the database are committed. The remaining three classes relate to monitoring actions on the database. The simplest of these is `threadLockExistsEvent` which uses the name of an object to conditionally acquire the lock when that objects exists in the database. This is a simplified and more efficient version of using an object of class `threadLockExpressionEvent` constructed from the expression

```
exists("<name of object>")
```

Instead, the C routine that implements the `getLock()` function can rapidly test the validity of this condition rather than employing the associated S evaluator.

In certain situations, it is convenient to watch for any assignments to a particular object or group of objects. We do this using an object of class `threadLockAssignEvent`. One simple but powerful example of this is to allow one thread to make changes to a variable and have other threads react to this change. This is a very simple of contraint-based programming and can be useful in graphical interface development. For example, consider a slider widget associated with a parameter that controls some aspect of a plot such as the bandwidth of a smoother or the ridge regression parameter in a linear model. As the user interacts with the slider, interface events are received by the event loop and the appropriate S expression invoked to update the plot. However, specifying a value for the object in any component of the interface must also be associated with suitable expressions to update the components that depend on this object. As we add more of these components, it can be a challenging programming task to manage such interaction. Instead, we can arrange for different threads to monitor assignments to this object and perform the appropriate action. This is effectively allows assignments to be treated as events and separates the assignment action from the update actions.

A second simple example of how we can use `threadLockEvents` involves monitoring iterative computations. Consider the EM algorithm described in 1 which imputes missing values in a dataset using a parametric model. At the end of each iteration, the worker thread might assign the intermediate results (parameter estimates and imputed values) to a `threadLock`. An ancillary thread waits for notification of assignment events to this database and then updates a plot displaying the history of the iterations and potential convergence. This illustrates how two threads can work in parallel with one monitoring the output from the other. The main work thread need not be aware of the existence of the other thread and can function whether it is present or not.

In addition to monitoring assignments, we may wish to monitor removal of objects from a `threadLock` database. This is done with a `threadLockRemoveEvent` object. The constructor functions `threadLockAssignEvent()` and `threadLockRemoveEvent()` for the two classes accept an arbitrary number of object names. All assignments are reported to a thread waiting on a `threadLockAssignEvent` object and it is the responsibilty of that thread to determine if the event refers to an object in the list of objects. The corresponding removal events are passed to a thread blocked on a `threadLockRemoveEvent`. Each event reported to these threads is itself an object of the corresponding class being waited on. The list of names in these objects contains a single entry which is the name of the object to which the action was applied. In the case of either class, specifying no names in the constructor function indicates that we are interested in any actions and that the match should be considered trivially valid.

We return to the extension of the class `threadLockExpressionEvent` that includes the names of the objects involved in the expression. We can implement this extension by adding another slot to the class which stores the names of the obects of interest in the expression. As with the implementations of the `threadLockAssignEvent` and `threadLockRemoveEvent` classes, a thread waiting on an object of class `threadLockExpressionEvent` would receive notification of all assignments events. For each event, it would match the name of the object being assigned against the elements in this list of names and evaluate the expression only if such a match existed. Thus, this `threadLockExpressionEvent` class can be implemented in terms of the `threadLockAssignEvent` and avoids unecessary evaluations of the expression. The identification of the relevant names in the expression can be automated in S since the expression can be decomposed. See 4.3.

3.2.3 Higher Level Synchronization Mechanisms

We have seen that a set of critical sections consists of blocks of expressions such that only one of a collection of threads can be executing within one of these blocks at any particular time. This ensures the sub-tasks are serialized, but not in any order, and that the shared data is protected across threads. There are situations where we would like to have a little more control over this serialization and let a subset of these threads operate within certain critical sections concurrently while excluding other threads and critical sections. For example, suppose we have one thread that repeatedly reads data from a device and appends it to a particular object. Several other threads are created to display different attributes of this data and perhaps perform sequential analysis as the data is gathered. For example, one thread may display histograms of the individual variables and a second pairwise scatterplots. A third thread may be updating a linear fit which is used to ensure the machine is properly calibrated and a fourth thread updating the decision regions of a sequential probability ratio test determining whether the process generating the data is stable or requires corrective intervention. Only, the thread reading and appending each observation modifies the object which is shared by all threads. The other threads treat the data as a read-only object. We might think of the input thread as a writer or producer and the other threads as readers or consumers. The set of consumer threads can access the object concurrently since none of these modify it and no race conditions exist among these tasks. However, none of these may be accessing the data while the producer thread updates the object and similarly the writer must ensure no reader is active within a critical section. Given the synchronization primitives intoduced to this point, all of the threads must acquire an exclusive lock before accessing the data and serialize access even for the consumer threads. For all the reasons described in 1, it is beneficial to allow concurrent access where possible.

This example illustrates that what is needed is a non-exclusive lock that permits concurrency for certain critical sections and excludes others. For these other critical sections, exclusive access is required. Such a lock is commonly referred to as a *reader-writer* lock [25]. The idea is that all reader critical sections can be executing concurrently but that writer critical sections which modify shared objects must wait to acquire an exclusive lock so that only one writer thread and no reader threads are executing at that point. There may be more than one writer thread and critical section, but each must acquire an exclusive lock.

By the choice of language in the paragraphs above, it should be clear that a reader-writer lock is an extension of `threadLock` and can be implemented using that class. We present an implementation of the reader-writer lock and methods which are based on the earlier synchronization primitives in the next few paragraphs.

The S class `readerWriterLock` is a simple extension to the `threadLock` class. It adds no extra slots as the data which maintains the state of the reader-writer lock is stored in the `threadLock` database.

```
setClass("readerWriterLock", representation("threadLock"))
```

The idea behind this implentation is simple. We use two counter objects to record the number of threads currently active. One counter is for the reader threads and the second is for the writers threads. The latter of course cannot exceed 1 by definition of the reader-writer lock behavior. The lock must be acquired while incrementing or decrementing the values of these objects but is not held during a critical section.

The constructor function creates the count objects for recording the number of both readers and writers.

```
readerWriterLock =
function(l) {
  l = new("readerWriterLock")
  as(l,"threadLock") = threadLock()
  assign("readers", 0, where=as(l, "threadLock"))
  assign("writers", 0, where=as(l, "threadLock"))
return(l)
}
```

Acquiring the lock for a reader involves ensuring that there are no writers currently active. The function `getReaderLock()` returns when the lock has been acquired. It waits until there are no writers holding the lock by using `getLock()` waiting on the condition that the number of writers is 0.

```
getReaderLock =
function(l) {
 getLock(l, Quote(writers==0)) # Wait until there are no writers
  assign("readers", get("readers",where=as(l,"threadLock")) +1,
                              where=as(l,"threadLock") )
 yieldLock(l)
}
```

```
yieldReaderLock =
function(l) {
 getLock(l)
   assign("readers", get("readers",where=as(l,"threadLock")) -1,
                              where=as(l,"threadLock"))
 yieldLock(l)
}
```

The corresponding functions for a writer lock are very similar. In acquiring the lock, the thread must check that no other writers are currently active and additionally that no readers are in a critical section.

```
getReaderLock =
function(l) {
   # Wait until there are no writers
 getLock(l, Quote(writers==0 && readers == 0))
  assign("readers", get("readers",where=as(l,"threadLock"))+1,
                               where=as(l,"threadLock"))
 yieldLock(l)
}
```

Releasing the writer lock simply involves assigning the value 0 to the `writers` object in the `threadLock`. The code for these functions could be easily consolidated to handle readers and writers together. This was not done here for illustrative purposes.

An alternative representation for the class would store the counts in the object itself. The `threadLock` object would be used to provide the synchronization between threads to the two count slots in the object as an exclusive lock. This would simplify the appearance of the code since the `get()` and `assign()` could be removed and the slots accessed directly. However, due to our implementation of the the `getLock()` conditional expression, the object must be stored in the database to trigger notification for the `getLock()` conditional wait. This could be done using `threadLockEvent` objects and monitoring any assignment to a dummy variable but this would confuse the example without changing the performance or functionality.

Barriers

Another commonly encountered synchronization idiom involves several tasks "rendesvouzing" at a common point before continuing with the remainder of their task. The idea is that each task evaluated in a collection of threads is divided into two or more logically different sections. At the end of the first section, each thread must wait for each of the other threads to have reached the end of its first section before advancing. This synchronization technique is called a *barrier* [25]. This is similar to `join()` but operates within tasks. In some sense these are like critical sections but are used to ensure that no thread advances out of the critical section before other threads have completed theirs. A common paradigm in which this is used involves iterative procedures. Consider iterative optimization using quasi-Newton methods. The goal is to minimize the mathematical function $f(x)$ which can be approximated by the quadratic form (from its Taylor series expansion)

$$f(x) \approx \frac{1}{2}xAx - bx,$$

where A is an $n \times n$ matrix and b an n-dimensional vector. The solution involves iterating until convergence is achieved, for example when the difference in the elements of x between successive steps (i and $i+1$) is "small". In other words, the optimal value of x is that for which

$$x_{i+1} - x_i = H_{i+1}(\nabla f_{i+1} - \nabla f_i) \approx 0$$

where H_i is the value of the Hessian matrix of second derivatives

$$\frac{\delta^2 f(x)}{\delta x_i \delta x_j}$$

evaluated at x_i. See [35] for details. At step $i+1$ in the iteration, the Hessian is updated based on H_i and x_{i+1} and x_i.

In certain circumstances, simply computing the elements of this matrix can be prohibitively expensive. However, these computations can be done in parallel and so can benefit from multiple processors. We might create n^2 threads each of which would iteratively compute the value of a single element. Before each thread proceeds with the next iteration, it must wait for the other $n^2 - 1$ threads to also have finished the i^{th} iteration also. Then the $i+1^{th}$ iteration proceeds with a consitent Hessian matrix. It would do so by blocking on a barrier which returns only when all the threads in the barrier have rendezvoused at the end of their subtasks. The code will be discussed below. [3]

Other procedures that might use a barrier include the EM algorithm [15] and simulated annealing [35]. Each would ensure that each child would wait for the other threads in the process to complete each stage of the algorithm before moving on to the next stage.

The barrier is a simple extension of `threadLock`. It is initialized with the total number of threads that must register with it before all of the threads can proceed. Additionally, it keeps a running count of the actual number that have registered with it at the current time.

```
setClass("Barrier",representation("threadLock", total="integer"))
```

We keep the total number of threads in the object itself and the running count in the `threadLock` database for the same reasons mentioned regarding the counts in the reader-writer lock class. The total number is fixed and the running count is dynamic and triggers `threadLock` events.

A `barrier` object is created using the constructor function

```
barrier = function(n) {
  b = new("barrier")
  as(b,"threadLock") = threadLock()
  b@total = n
```

[3]We have clearly ignored the details of updating x and checking for convergence. This would probably be done in the master thread using the n^2 threads as slaves simply to compute H_{i+1}. Each of these threads would notify the master and then block on the barrier. This is not necessarily a good implementation of this iterative procedure but is used simply to illustrate how a barrier works and the type of situation for which it might prove useful.

```
  assign("count", 0, where=as(b,"threadLock"))
 return(b)
}
```

Each thread waits at the barrier by calling wait() with the barrier as the single argument.

```
barrierWait =  function(b) {
  getLock(b)
   tmp = get("count",where=as(b,"threadLock"))+1
   assign("count", tmp, where=as(b,"threadLock"))
   getLock(b, condition = Quote(count == b@total))
  yieldLock(b)  # give up the lock and proceed

 return(T)
}
setMethod("wait", signature("barrier"), barrierWait)
```

Each of the n^2 threads in the example above might have a task of the following form. The task iterates indefinitely until the master thread determines that the computation has converged. The value of the (i, j) element of the Hessian is computed outside of a lock and the value assigned within a lock. The expression Symbolic.Hessian[i,j] is assumed to refer to the objects x, etc in some suitable manner. The iteration then signals the master thread that it has completed by incrementing the value of threadCount. Then it waits until all the threads have completed this iteration including the master thread.

```
while(!converged) {  # converged common to all, but protected by
                     # barrier since only set in the master thread.
   tmp = eval(Symbolic.Hessian[i,j])
   getLock(hLock)
   h[i,j] = tmp
   yieldLock(hLock)

     # signal the master thread that it should consider the convergence
   getLock(masterLock)
     assign("threadCount", get("threadCount",where=masterLock)+1,
                                              where=masterLock)
   yieldLock(masterLock)

   wait(b)
```

```
}
```

The master thread resets `threadCount` to 0 at the end of each iteration and initializes the barrier total to $k+1$ where k is the number of children threads.

Semaphores

A commonly used idiom in the examples above has been one thread waiting on several others to complete a task or sub-task. An integer object is created to store the count of the completed threads and one thread uses the

```
getLock(l, Quote(count != k))
```

conditional wait idiom to test and acquire the lock. As each thread finishes, they increment the count using something like the following:

```
getLock(l)
  assign("threadCount", get("threadCount",where=l)+1,where=l)
yieldLock(l)
```

This procedure can be replaced using a higher level interface which provides protected access to a counting variable and allows for simple testing of its value. Such counting objects are usually termed semaphores [19] [25]. The terminology is that one *waits* on a sempahore for the count to be equal to some specified value and other threads *post* the sempahore which increments its count. The implementation of such semaphores follows from the examples above and makes using the synchronization paradigms a little easier.

Task Queue and Synchronization

The C-level thread interfaces do not directly provide routines or structures to allow threads to communicate with messages. Instead, since threads share a common address space, communication is done directly using shared objects and left to the programmer. This is done using *mutex* and *condition variable* objects. The thread model we have implemented offers three communication mechanisms. The first is the same as the POSIX thread model using `threadLock` objects and condition expressions. The second method is the indirect messaging performed by the evaluator manager. This greatly simplifies the C level implementation as no thread has direct access to another thread's internal structures. Instead, requests are serialized in the manager avoiding some synchronization issues between threads.

The final mechanism, and the focus of this short section, offers what might be considered a more intuitive approach to synchronization. This mechanism is the `sendTask()` function that allows one thread to "control" another thread without having direct access to its variables. The same principle of serializing requests within a thread on behalf of another, as used in the evaluator manager, simplifies the synchronization. Of itself, this is a simple mechanism that offers little more than a more structured interface to communication via messages in the form of S expressions. However, when coupled with the feature that threads (and their databases) persist across tasks, this communication mechanism offers a different approach to synchronization not directly available in other languages. We consider how we might use this mechanism to implement the barrier concept introduce above.

The barrier can be thought of as having k threads with each thread associated with a task. A parent thread, t_0, creates child threads $t_1, \ldots, t_k$. Using the barrier as developed above, the parent passes each thread t_i an expression which contains calls involving the barrier object. An alternative approach is to consider each of the tasks as two subtasks — the pre-barrier task and the post-barrier task. We denote these as e_0^i and e_1^i for the i^{th} thread. In this setup, the parent starts by creating each thread i with task e_0^i. Then it uses `join()`s to wait on all of these pre-barrier tasks to complete. Then it sends the tasks e_1^i to the respective threads to complete the overall task.

```
th = vector("list",k)
pre = vector("list",k)

for(i in 1:k) {
    th[[i]] = thread(substitute(foo(i)), list(i=i))
}

join(th) # wait for the first set of tasks to complete

 # send the next set of tasks.
for(i in 1:k) {
    th[[i]] = thread(substitute(bar(i),list(i=i)))
}
```

Let us return to the iterative optimization. Rather than arranging for each slave thread to be evaluating the element of the hessian multiple times using a loop, the master thread sends the thread that task in each of its own iterations. In other words, we are replacing a single task with multiple iterations into a sequence of tasks for which each element corresponds to an iteration. Then it joins on all the threads, proceeding to update x and test for convergence only when all of the n^2 slave-threads are finished. With this approach, the parent thread's task migh look something like the following.

```
 # initial values x.prime and for Symbolic.Hessian,
 # and n are assumed to be created somewhere else.
```

```
hLock = threadLock()
h = matrix(0,n,n)
h.prime = matrix(0,n,n)
threads = vector("list", n)

  # create the n*n threads as idle threads
for(i in 1:n) {
 threads[[i]] = vector("list",n)
 for(j in 1:n) {
  threads[[i]][[j]] = thread()
 }
}

while(!converged) {
x = x.prime
h = h.prime

 # now send each thread a task for this iteration
for(i in 1:n) {
 for(j in 1:n) {
   sendTask(substitute({
                     tmp = eval(Symbolic.Hessian[i,j]);
                     getLock(hLock);
                     h.prime[i,j] = tmp;
                     yieldLock(hLock)
                    }, list(i=i,j=j, hLock=hLock)),
        thread= threads[[i]][[j]])
 }

 join(unlist(threads,recursive=F))

 update(x)  # update x.prime based on the new value of the hessian
            # in h.prime

 converged = sum(abs(x.prime - x)) < tolerance
}
```

Note that this approach is both conceptually simpler and also removes the need for each slave thread to notify the master that it has completed its task so that the master can update and test for convergence. This messaging scheme since based on a powerful language — S, is a very useful synchronization approach. It will perhaps suffer from a slight loss in performance but we will not discuss this here.

3.3 Manager Functions

Since the evaluator manager has access to all the internal structures that make up the S engine, it is capable of examining the contents of these and determining the state of the environment or some subset of threads. The `lockOwner()` requires the caller to identify the `threadLock` object. We introduce the atomic function `LockOwners()` in order to be able to retrieve a list of all the locks currently in existence and the thread object that currently holds each lock. This allows one to examine thread deadlock or situations in which data is corrupted due to an absence of locks. Similarly, with a minor modification to the C code and a slight performance penalty, the evaluator manager can identify which threads are blocked waiting to acquire all the `threadLock` objects. These tools allow the user to view snapshots of the environment at almost any time and so debug complex synchronization problems. Each will be added as environment stabilizes and is tested more extensively in its current state and provided as a runtime option to modify the behavior of the evaluator manager.

Chapter 4

Examples

As we mentioned in chapter 1, threads can be used to access multiple processors yielding fast parallel computations and also to facilitate the development of concurrent but independent features in an environment, such as those described in section 4.2. In this chapter, we look at such examples arising in statistics, starting with numerical computations that might access multiple CPUs. Here we illustrate some of the details of implementing the examples mentioned in 1.4. Our focus will then turn to examples related more to tools for the user's environment such as help facilities, data servers, event loop applications, etc. and how these are easily integrated into an existing threaded environment. Finally, we outline the topic of automated threading and how it might be performed in S. This illustrates how S can be used in the development cycle as a useful prototyping tool for threaded applications or directly in interactive applications.

Throughout this chapter, our intention is not to present complete or ideal implementations of particular problems. Instead, we wish to illustrate the types of problems to which this work can be applied and also illustrate the aspects of threading that might not yet be clear to the reader, particularly synchronization. Certain examples are not implemented the way we would necessarily choose in a production quality product and the code shown here should not be taken as suggested implementations.

4.1 Computational Examples

We examine three statistical computational procedures in which relatively straightforward implementations using threads are developed. The main reason for selecting these is that they are relatively simple and commonly known procedures and so do not require the reader to concentrate on the mechanics of the procedure. Instead, the focus is on the use of the thread functions to implement the top-level structure of the code. The examples illustrate that utilizing threads demands slight extensions to a non-threaded implementation. These extensions are concerned with managing the threads and synchronization.

The three examples are cross validation, bootstrapping and image registration. Additionally, we also outline how threads can be *easily* used to monitor the state of a computation via intermediate values. We do not focus here on a variety of parallel algorithms that are very different from their non-threaded counterparts while still computing the same values. These are typically non-intuitive and remove the focus from the tools to the algorithm. The examples developed below are relatively general and can be used as templates for the type of threaded applications we feel the tools are best suited. It is our contention that modifying existing algorithms to work in parallel is relatively straightforward with the tools we have developed and an understanding of the nature of thread synchronization conflicts.

The code that implements the examples will run on both single and multi-processor machines. On a single processor machine, a threaded implementation might be fractionally slower than a corresponding non-threaded implementation due to the extra steps involved in managing the threads. As we mentioned earlier in chapter 1, if the threads are I/O bound, the opposite might be true, but the the differences in performance will be small in either context. The main benefit arising from a threaded implementation is the increase in performance attained on a multiple processor machines. While optimal performance may require carefully tailoring the number of threads to the available number of processors, etc. a large performance improvement should be realized with no such non-portable modifications.

One of the classic uses and examples of parallel computation is matrix manipulation. The examples we present below and many others that arise in statistical computations do not fit naturally into this category of parallelizable problems. While the functionality of the thread extension introduced in the previous two chapters does allow parallel matrix manipulation and other similar computations, the functional nature of the S language is not best suited to these scenarios. These computations are characterized by element-wise assignments into a larger object. For example, a parallel matrix multiplication algorithm might have a thread compute the element

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Since S has no pointers or references, the assignment to the element `c[i,j]` must be executed in a critical section that protects the entire object `c`. If the computation time of the right hand side of the assignment is not large relative the time for the acquisition and release of the lock, any performance benefit attainable by parallelization may be lost as the threads effectively become serialized in terms of access to the object `c`. As a result, in the examples below and as a general rule of thumb, it may be best to have each thread return the value of its computation to the master thread (via a call to `join()`) and have the latter arrange these elements into an appropriate structure (such as a matrix, etc.)

The most common computations that arise in a statistical context involve summations and repeated evaluations of the same function with different arguments and a mixture of the two. These characteristics lend themselves to a multi-threaded approach and are visible in the examples below.

4.1.1 Cross Validation & The Bootstrap

Cross-validation is a useful statistical technique employed to compute prediction parameter estimates in a non-formula based manner. The procedure involves creating k disjoint subsets of the data by randomly arranging the n observations into k groups and for each of these groups removing these observations from the original dataset. For each of these k subsets, the parameters of the original model are re-estimated and the observations not used in this fit are predicted using the new estimates. These predictions can be combined to give a more realistic estimate of the prediction error of the model than the standard formulae. See [16], chapter 17 for more details.

The standard bootstrap ([16], chapter 6) is a procedure in the same spirit as cross validation as it uses the observed data to produce non-formula derived parameter estimates. The basic idea is that there is a population distribution for the random variable[1] X denoted by F_X from which the n observations, $x_1, \ldots, x_n$, are sampled. This can be estimated using the empirical distribution, $\hat{F}_{X|x_1,\ldots,x_n}(x)$. The bootstrap procedure involves treating $\hat{F}_{X|x_1,\ldots,x_n}$ analogously to F_X, and repeatedly sampling from it (with replacement) to produce B samples of size n, denoted by $\underline{\mathrm{x}}_1^*, \ldots, \underline{\mathrm{x}}_B^*$. The statistic of interest, say $T(\underline{\mathrm{X}})$, is computed for each each of these new samples and its empirical distribution computed. We denote this as $\hat{F}_{T|T_1^*,\ldots,T_B^*}(t) = \hat{F}_{T|T(\underline{\mathrm{X}}_1^*),\ldots,T(\underline{\mathrm{X}}_B^*)}(t)$. Under certain conditions, this distribution converges to the distribution $F_T(t)$ and can be used to estimate different parameters of interest such as quantiles, variance, etc.

For computational purposes, the two procedures share one important feature. Each involves a fixed number of operations or sub-procedures in which a subset of the original observations are used to fit a particular model. From our perspective, each of these operations is a separate task and can be performed independently of the others. As a result, we can evaluate each in a separate thread and potentially do the entire computation in parallel.

Below, we develop relatively general functions for each of the procedures. We start with cross validation. This takes a data frame and a function which performs the computations within an iteration. Additionally, the number of groups into which the observations are to be split (k) is specifiable with a default of 10-fold cross validation. The function creates k threads and waits for these to terminate using `join()`. Rather than passing the inputs directly to the threads, the parent thread arranges to create a temporary database which can be used to store these and is visible to each of the children threads. The task expressions refer to these and the objects are resolved when evaluated rather than when the task is created. This database also acts as a synchronization

[1]We use a 1-dimensional or scalar random variable to simplify the notation. The same approach works for multivariate random variables

lock for the results object. Rather than building knowledge of the the object to which the results should be assigned into each task, we use `join()` to retrieve each threads return value and process these in the master or parent thread.

```
CrossValidate = function(data, fn=CVFit, k=10)  {
   # create threadlock to allow the threads synchronized access
   # to the results vector.
 l = threadLock()
 attach(l,pos=1)  # threads will see the objects in this

 n = row(data)
 assign("data", data, where=1)

 num.iterations = k

 threads = vector("list",num.iterations)
 for(i in number.iterations) {
   threads[[i]] = thread(substitute(foo(i,n,k),
                              list(i=i, foo=fn, n=n, k=k)))
 }
 results = unlist(join(threads))
 results = get("results",where=l)
 detach(l)
 return(list(MSPE=mean(results^2), Errors = results))
}
```

The function which performs the fit can treat the input data as fixed and need not acquire a lock on it. However, a reader-writer lock as developed earlier might suitable to ensure synchronized access while allowing all threads to execute in parallel. It computes the indices of the observations to use for fitting the data and then computes the parameters. Next, the prediction errors are computed.

```
CVFit = function(which, N, k) {
  start =  i*N/k
  end  = (i+1)*N/k
  output = start:end
   # perhaps we might use a reader-writer lock
   # to ensure correct access to data
  input = data[-output,]
```

```
    fit = lm(y ~ ., as.data.frame(input))

      # Compute the prediction errors
    errors = (data[output] - predict(fit, data[output,]))^2
   return(errors)
  }
```

If we had chosen to assign the results as part of the task rather than have the parent thread gather them, we would add the following expressions to the end of the task.

```
      # and assign these to the results vector.
      # Make sure this is locked.
    getLock(1)
     results = get("results", where=1)
     results[output] = errors
     assign("results",results,where=1)
    yieldLock(1)
```

Additionally, the parent thread would create the `results` vector before creating the threads.

Implementation of the bootstrap procedure is similar. A thread can be created for each of the `B` samples. The function used to compute the statistic from this sample is supplied as an argument. Since the class of the output object of this function is unknown, this generic function merely gathers the return values from each thread. It does this with a call to `join()`.

```
Bootstrap = function(data, T=mean, N) {
  l = threadLock()
  group = threadGroup()
  attach(l, pos=1)
  assign("data", data, where=1)
  for(i in 1:N) {
       # returns threadTask
    tmp = thread(substitute(BootstrapSample(T),list(T=T)))
    group = add(group, tmp)
  }
  vals = join(group)
  return(vals)
}
```

The function which produces the sample and statistic, `BootstrapSample()`, is relatively simple and is shown at the end of the section.

Cross-validation typically involves dividing the observations into a relatively small number of groups — on the order of 10 or more. Creating a thread for each group is a reasonable strategy. The bootstrap is different however. In many cases, the number of samples required to construct a reasonable estimate of the empirical distribution of the statistic T is between 100 and 1000. Creating a thread for each sample is not a viable strategy as each thread consumes memory (for its S- and C-level stacks) and also must be scheduled. There is a (configurable) maximum number of threads that can be supported on any system and this depends on the machines configuration. However, the most limiting factor is the number of processors. We would like each thread to run on its own processor since the scheduling of more threads consumes resources. Thus, for the bootstrap, we cannot use the above implementation with much success. Instead, we will arrange for a small number of threads to perform multiple samples. This can be done in a several ways. For each thread, we create one task that performs multiple samples and returns a list of the results. Alternatively, we could send each thread multiple tasks and arrange for each of these to assign the results into a common object in the `threadLock`. While simple, this approach is slightly more complicated than the former as the master thread must join on the final task of each thread and each task must acquire the lock.

The following is an implementation of the first approach.

```
Bootstrap = function(data, T=mean, N, k = 20) {
  l = threadLock()
  attach(l, pos=1)
  assign("data", data, where=1)

  group = threadGroup()
  for(i in 1:k) {
      # compute the number of samples to be done by this thread.
      # only the last one will differ.
    n = as.integer(N/k)
    if(i == k)
     n = N-k*n
    tmp = thread(substitute({
                     ans = vector("list",n)
                     for(i in 1:n) {
                       ans[[i]] = BootstrapSample(T)
                     }
                      return(ans)
                     },
                     list(n=n,T=T)))
```

```
        group = add(group, tmp)
      }

      on.exit(detach(1)) # throw away threadLock database.
      vals = join(group)
      return(vals)
    }
```

In both cases, the function that produces the sample and computes the statistic is the same.

```
    BootstrapSample = function(T) {
        which = sample(1:nrow(data))
        vals = T(data[which,])
      return(vals)
    }
```

Note how we dynamically create the task expression above to call this function rather than modifying the function to take different arguments.

4.1.2 Image matching via correlation

We consider an image registration problem in which 2 images, I_1 and I_2, are available — each in the form of a matrix. We denote the dimensions of these images by (m_1, n_1) and (m_2, n_2). The problem is to determine the optimal position $\hat{p} = (\hat{x}, \hat{y})$ for which the images are, in some sense, most similar. In this example, we are assuming that the images are recorded on the same scale so that the search is for a simple translation rather than a scaled or non-affine transformation. The similarity measure we use here is computed by the function `m()` which computes some similarity value for the intersection of the images. For example, this might be the simple correlation function which ignores the spatial structure of the pixels in the image or a simple count of the number of positions (x, y) for which the values are identical. The measure m is assumed to be symmetric in its arguments, I_1 and I_2.

We write $I'_k(p)$ to denote the image I_k translated by $p = (\Delta x, \Delta y) = (p_x, p_y)$. That is

$$(I'(p)) = (I'_{i,j}) = (I_{i+p_x, j+p_y})$$

We are searching for $\hat{p}$ which satisfies

$$\max_p m(I_1, I'_2) = \max_p m(I_1, I_2(p)), \text{where } I_2(p) = (I_{i+p_x, j+p_y})$$

The procedure to determine $\hat{p}$ involves exhaustively searching all possible values p such that there is a non-empty intersection of the two images. We might insist that the area of overlap is greater that some minimum number of pixels or area so that the estimate of the measure of similarity is somewhat meaningful.

The naive approach as with the bootstrap is to create a thread for each point p for which the two images I_1 and I_2' have an non-empty intersection. For reasonably sized images, this would create too many threads so instead we create a small number of threads and perform multiple searches within each of these threads. For the sake of illustration, to contrast the bootstrap function above we will implement these multiple searches as separate tasks within each thread rather than as one single task. For simplicity, we will assume that we can create as many threads as there are possible positions for p_y $(n1 + n2 - 2)$ so that the images overlap. This simply avoids unnecessary details in the S code shown below.

```
registration = function(I1, I2) {
 l = threadLock()
 attach(l,pos=1)

   # create a matrix in which the element-wise results are stored for
   # each (x,y) point.
 assign("vals", matrix(0, (m1+m2-2), (n1+n2-2)), where=1)

 m1 = dim(I1)[1]; n1 = dim(I1)[2]
 m2 = dim(I2)[1]; n2 = dim(I2)[2]
 num.columns = n1+n2-2

   # create the threads with no tasks.
 threads = vector("list",m1+m2-2)
 for(i in 1:length(threads)) {
  threads[[i]] = thread()
 }


 group = threadGroup()

 for(i in 1:length(threads)) {
   for(j in 1:num.columns) {
      th = sendTask(substitute({getLock(1);
                                vals[i,j] = m(i-m, j-n);
                                yieldLock(1);
                               }), list(i=i,j=j,m=m,n=n),
```

```
                                      thread = threads[[i]])
          if(j==num.columns) {
              # store the last task for this
              # thread for use in a call to join
            group = add(group,th)
          }
       }
     }

     join(group)  # wait on these tasks to be finished.
       ...        # compute the optimal value
     return(vals)
    }
```

The function stores the results matrix in a `threadLock` database. The similarity measures for each offset $p = (x, y)$ will be assigned to the corresponding entry of this matrix. Each thread created to compute the similarity measures for a given row of positions acquires the database lock during the assignment in order to make the assignment action atomic. The master thread waits for all the tasks to complete with a call to `join` based on the final task for each thread.

4.1.3 Multiple Threads for a Single Task

Another situation in which parallel processing can be useful is in the search for a solution to a problem for which there are several possible approaches or algorithms. The algorithms can be radically different but the focus is on the output from any of them and not the comparison of the procedures. An obvious example occurs in maximizing a likelihood function. Different algorithms exist for optimizing a mathematical expression and each has different properties that make it suitable for use. When equipped with multiple processors, we can employ each algorithm to solve the optimization problem in parallel and wait until one returns with a solution. A similar type of problem involves using multiple starting points as inputs to the same algorithm. We assume we have a unique solution that can be determined by this algorithm. A thread can be created for each starting position and the overall procedure waits for any one of them to finish and report *the* result. By having multiple starting points, the hope is to find one that causes the algorithm to converge quickly. By creating multiple threads executing in parallel, we have the luxury of having the threads simultaneously seek the solution without consuming resources from another thread and potentially reduce the overall time to determine the solution.

We consider the multiple starting points problem here. The procedure that implements the algorithm is called via the function `foo()`. We establish the threads in the usual way. Each task assigns the value returned by the call to `foo()`. This signals the master thread which waits for the existence of this object. This thread then terminates the other threads which are still actively searching for a solution. This is done via the `threadStatus()` function.

```
threads = vector("list",length(points))
l = threadLock()

for(i in points) {
  e = substitute({tmp = foo(point); assign("value",tmp,where=l),
                  list(point=i,l=l))
  threads[[i]] = thread(e)
}
getLock(l, "value") # wait until the object "value" is assigned
threadStatus(threads,"TERMINATE")
value = get("value", where=l)
detach(l)
```

While it is good practice to acquire the lock before committing assignments of shared objects, this situation does not absolutely require the worker threads to do so. The reason for this is that we do not care which value we get as long it is a solution from one of the methods. The database assignment action ensures that the lock is held for the duration of that action. At this point, either the master thread wakes and acquires the lock or another worker thread starts the assignment. In either case, the master thread can only acquire the lock when a worker thread has implicitly released it after making the assignment. However, explicitly acquiring the lock does not change the computations and would make it clearer.

4.1.4 Monitoring Computations

The threaded environment has the potential to make functions more modular and allow them to be connected in interesting ways. Consider a function that is divided into multiple steps. For example, statistical procedures that require inputs to be filtered through different stages in the procedure and computations that loop over values such as summations are examples of such tasks. In order to manipulate the intermediate results, (e.g. graphically displaying them in the form of a plot) the function or task must include the appropriate commands. If this behavior is not desired in all circumstances, additional arguments must be added to the function, complicating the user interface.

The threaded environment allows these types of overall tasks to be divided into subtasks, each self contained and reusable. The computations are done in one task and the other tasks wait for the intermediate values to be assigned to a shared database. These threads are notified of these assignments via a `threadLockEvent` object and then display the values appropriately.

Consider for example a thread whose task is to iterate a particular computation We have already seen the quasi-Newton optimization routine in 3.2.3. Other examples include the EM algorithm, smoothing parameter estimation, etc. The functions that implement these algorithms can be made to assign their intermediate output values to a `threadLock` database by simply attaching the `threadLock` as the working database for this thread and explicitly assigning the values there rather than locally in the evaluation frame. The behavior of the computations is unchanged as the assignments are still done to memory databases rather than to disk.

An outline of the code for this example is given below. We use two threads — a worker thread that performs the iterative procedure and a second thread which monitors the assignments and updates the display. This monitor thread sits in an infinite loop

```
attach(l, pos=1)
work.thread = thread(Quote(iterate(data, starting.values)))
display.thread = thread(
  Quote({
           while(T) {
               # wait for an assignment to the object theta
            getLock(1,threadLockAssignEvent("theta"))
               # get a local copy to allow release of the lock
             tmp = get("theta", where=1)
            yieldLock(1)
             display(tmp) # display this value.
           }
         }))

  join(work.thread)
  threadStatus(list(display.thread,work.thread), "TERMINATE")
```

The function used in the work thread is then defined in the following manner.

```
iterate = function(data, starting.values) {
  converged = F
  old.theta = starting.values
  while(!converged) {
   theta = foo(data, old.theta) # compute next iteration's starting values
   assign("theta", theta,where=l) # notifies other threads of
                                  # intermediate values
   if(isConverged(theta))
```

```
        converged = T
      old.theta = theta
    }

    return(theta)
  }
```

The only modification necessary to fit into this framework is the assignment to the `threadLock` database. This is responsible for triggering the notification of the monitoring threads.

The benefits of this approach include the simplicity of each task and easier maintenance of the code. Each task can be developed independently of the others and allows for different features to be added in the future. Additionally, we are not restricted to having just a single thread monitor the intermediate results. Just as in many event-driven environments, multiple tasks in different threads can wait for these assignments and work independently. For example, as the optimization starts to converge, the user may chose to terminate or suspend the thread, satisfied with simply knowing the general region of the solution. This would monitor interface events (rather than database events) and invoke the appropriate expressions in response to these. Note how easy it is to add and remove this functionality without modifying any of the tasks.

This style of monitoring also lends itself to interactive debugging. S uses the approach of modifying the function being debugged by inserting calls to interactive browsing functions at suitable points in the function and assigning it to frame 0. Doing so means that the modified function aliases the original function and is used in subsequent calls. The monitor style in which one thread views the contents of and controls another thread allows a more typical style of debugging common in compiled languages. The monitor thread can be established (with some modifications to the internal S engine) to control the behavior of the target thread and task.

4.2 System/Environment Applications

Rather than focusing exclusively on the traditional numerical aspect of statistical computing, version 4 of S allows more “system” and “environment” application oriented development. Four of the most important features in this newest version of S are: (1) a complete object-oriented approach through classes and methods treating everything as an object. (2) connections allowing structured communication with other processes. (3) internally embedded C level pointers and raw (non-S) data. (4) memory mapping for large objects. Examples of these environment applications which use these features include client-server models offering statistical and database functionality, graphical user interfaces (GUIs), real-time production and performance monitoring. These are features frequently developed and performed outside of the statistical domain in which they are used and to which their output may be filtered. The ability to develop these types of applications should allow statistical computing to assist in all aspects of data analysis including collection and presentation. With the addition of threads, their implementation should be greatly simplified and easier to understand. We describe three of these briefly here and illustrate how we might implement them, with emphasis on the relation to threads. The three topics are GUI development, a help mechanism and an object or data server.

4.2.1 GUI Applications

The development of user level threads in S grew out of another project the goal of which is to provide tools for developing graphical user interfaces (GUIs) in S. (This will be described in another document to be prepared in the near future as a Lucent Technologies technical report.) There are three main reasons for adding these tools:

- We wish to allow new statistical methodology developed using S to be used by practitioners unfamiliar with the S language.
- Integrate existing technology developed outside of the statistical domain into a statistical environment. This includes adding new data editing and visualization tools.
- Simplify many of the trivial tasks commonly performed by typing that are more easily performed by the point-and-click style interface. These include interfaces to the modeling and formulae module in S, object management across chapters, history selection mechanism, etc.
- Experiment with new ideas in data visualization and visual languages [8].

Consistent with the notion of S as an environment, the intention is to allow multiple GUI applications to be in existence simultaneously within a single S session and allow the user to be simultaneously executing different tasks in several of the applications. One of the primary motivations for threads in this context is to allow each S task associated with a user action to be performed in a separate thread from the event loop that monitors user actions. In the absence of this arrangement, the event loop detects user actions as events and invokes the task associated with the current event. Control is returned to the event loop only when this task has completed and only then is the next event and task processed. The event loop cannot respond to user actions for the duration of each task and if this task involves a lengthy computation (such as fitting a model to a large dataset, etc.) the latency is highly undesirable and can give the user the impression that the application has stopped. Indeed, the user has no mechanism to cancel a task via the GUI since the task associated with this cancellation is not handled until the target task has been completed. Additionally, we may want to have multiple tasks being evaluated concurrently such as user action tasks and a lengthy background task or a graphics thread that updates the display when necessary.

User level threads allow a relatively simple solution to this problem. The event loop which listens for events sent from an X server is implemented in one thread, thread A. We create

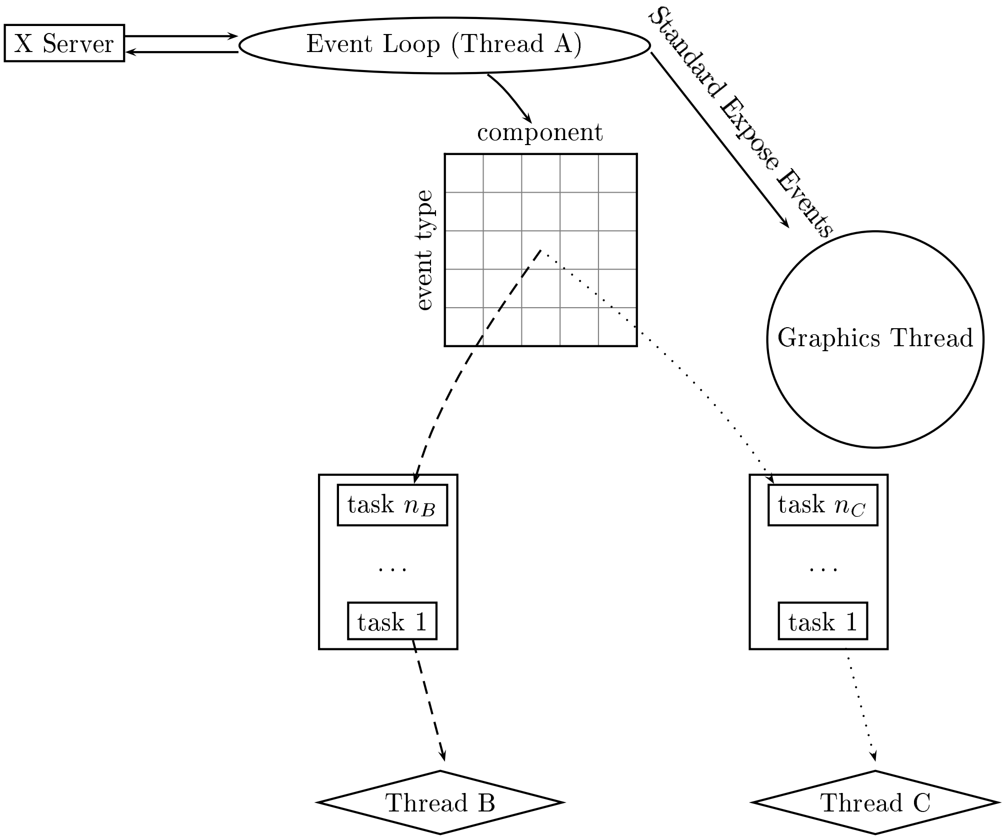

Figure 4.1: GUI Application Control Structure.

a second thread, thread B, which is to be used only for evaluating the tasks associated with user actions. When an event is received, the associated S expression is retrieved from a table (using the interface component and event type as the lookup keys). Then we use the `sendTask()` function to arrange for this task to be evaluated in thread B. Immediately after the task has been sent to thread B, thread A is available to respond to other events. Since this is communication of the task between the two threads is a simple computation, the GUI can respond to other user actions very rapidly. These can include cancel and undo commands which refer to task queued or being evaluated in these worker threads.

A more sophisticated version of this approach involves using more than one thread to evaluate the callback tasks and show in figure 4.1. Instead of creating a single thread to which the tasks are sent, we create multiple threads which are responsible for evaluating different groups of expressions. This allows several tasks within the GUI application to be executing concurrently.

This multi-threaded version of the GUI framework clearly offers greater utility to the user of the application. S/he need not wait for one task to finish before seeing the results of another. Also, one thread can be dedicated to tasks that must be constantly performed, such as rendering images on components on receipt of expose events or garbage collection, while the other threads can evaluate other tasks. This additional user utility comes at the price of increased programming complexity. Since the different event tasks are being evaluated concurrently rather than sequentially as in the first approach, the programmer must be concerned with synchronizing the tasks across threads. Additionally, it might be advantageous to associate groups of different task-event pairs with threads so that within-group tasks are serialized within one thread. We outline a methodology for partially automating the grouping of the tasks and the synchronization identification in section 4.3.

An initial attempt to introduce GUIs into S in the absence of threads involved making the event loop a reader. This forces the communication between the X server and client (the GUI) to be treated as an S connection. This is feasible since the connection is on a socket which shares its reliance on a file descriptor with S `connection` objects. On at least one occasion, changes to the way connections were handled caused all GUIs, including the default graphics device, to fail. Additionally, since all GUIs within a non-threaded S environment are run in this manner, controlled by the same global event loop and compete for CPU time, a GUI may become non-responsive as it is starved of resources. While this is also possible using threads, the granularity of the scheduling is much finer and so less likely. This experience illustrates the benefits of using threads for independent tasks. The internal implementation is greatly simplified as each of the event loops are now implemented in their own thread that blocks when the queue is empty. Aside from the synchronization issues, the development of a GUI is simpler and supports significant feature enhancements due to the presence of threads.

4.2.2 Help Daemon

In our initial work in the GUI environment described in the previous section, we have a help command that displays the relevant text from the S `help()` function in an HTML rendering widget[2]. While the conversion from the internal structure of the documentation to HTML is provided through S filters and is relatively efficient, it would be more efficient to perform the conversion once rather than each time the contents of the help document are displayed. Also, in many contexts, it is better to avoid having multiple help windows displaying the same topic. This might arise if two or more applications within the S environment provide help on the same topic. Instead, we would like to coordinate the help to show a single version across the applications. Additionally, we do not want each GUI developer to have to implement his/her own help display system in the interest of efficiency and interface consistency. Instead, we solve this problem by having one central help facility which caches the filtered documentation and manage the display for all applications within the environment. Essentially, this is a help daemon thread that acts as a server to all the application clients. These clients send requests to the help server for a particular

[2]This code is part of the the National Center for Supercomputing Applications (NCSA) Mosaic browser that is publically available.

topic and announce when they have finished with that display. The help server is in charge of creating and managing the windows and the documentation objects. It is a daemon thread in that it starts when first needed or at the start of the session and remains in the background servicing requests from all applications within the environment.

We might implement this using one globally visible thread whose job is to control the help facility. The thread object is stored in the session database using some pre-defined name, say `.HelpThread`. Thus it is retrievable by any thread via the expression

```
get('.HelpThread', where=0).
```

The communication mechanism between client and server can be arranged in either of two ways. The simples version is to use `sendTask()`. A client requesting help on the function `lm()` might post a request

```
sendTask(Quote(showHelp(lm)),get(".HelpThread",where=0))
```

The help thread then evaluates the call to `showHelp()` processing the request. Alternatively, the daemon thread can be created to use a different evaluator specified by the `action` argument. This might be a lightweight evaluator which accepts only strings identifying the topic and handles this in much the same way as above but allowing the clients to avoid creating the task expression.

A second approach that demands more work from the developer of the help system is to mimic the request/task queue we currently use to send task requests to threads. The idea is that we maintain a single queue or list of topics for the help thread. Clients append elements to this queue as requests identifying the topic. The help thread processes these these requests sequentially. This is described in more detail in chapter 5. For this to work, the help thread requires an associated `threadLock` database, again known to all. (We would store the help thread and `threadLock` objects as one object as `.HelpThread` in the session database.) Each client application sends a request to the help daemon by acquiring ownership of the `threadLock` object. Then, it appends the request (in this case, simply the topic string - e.g `"lm"`) to a list of previous requests. If the list does not exist, the thread creates it before appending the request. Finally, it relinquishes ownership of the lock. The assignment wakes the help thread signaling that the queue of requests is non-empty. Additionally, the release of the lock allows other client threads to compete for access to the queue.

There are two simple ways to organize the list. We might have one list and each thread inserts the topic and its own thread identifier `self()`. Alternatively, we might have a separate list for each thread stored in the database using the thread identifier as the name of the object.

The help daemon in this approach is a simple loop implemented by the following code.

```
while(T) {
  getLock(lock, condition = threadLockAssignEvent())
```

```
     obj = get(name, where=lock)
     assign(name, list(),where=lock)  # remove the topic from the list
    yieldLock(lock)

    for(i in obj) {
      showHelp(obj[i])
    }
}
```

Note that all the regular thread status commands are available to control the help thread. So we can suspend, terminate, etc. the daemon thread. Thus no specific controls need to be put in the daemon task to monitor such requests.

The three different approaches we have outlined here illustrate the flexibility of threads and how they simplify development of features within an environment.

4.2.3 Data server

S databases provide a convenient mechanism for storing data for use across tasks. The databases are either in memory or on a directly accessible file system. In some senses, a database can be considered a data or object server. Calls to `get()`, `exists()` and `assign()` are the message types used by a client (an evaluator) to evoke a response from the database or server. There are several situations in which it is convenient to extend the server concept to provide different or enhanced functionality. Another project I was involved in during the course of this work related to developing web based Java applications for teaching introductory statistics to university freshmen. In this project two situations arose in which the need for a threaded centralized database or object server was found to be a useful approach. This section outlines the details of the development of this object server and presents an implementation in S. We note also that a variation of the help server could be implemented in this manner to return previously converted documents to requesting threads.

The first situation arises commonly in graphical user interfaces. A GUI may consist of several windows (or views arranged as stacked cards within a window) but the user will only interact with one of these initially. Some of these views need never be created as the user will not access them during the life of the application. Moreover, they may be computationally expensive to create. Accordingly, if the user does request such a view, the delay between the request and the display of the view may be quite considerable. As a result, it might be appropriate to create these views at the start of the application in an effort to reduce run-time latency. Of course, performing these expensive computations at initialization increases the user's perceived delay for the application to start which is an important measure in the usability of an application [43]. So there is a tradeoff between startup and run-time latency.

In an attempt to lessen this tradeoff, we notice that a characteristic of graphical interfaces and event-driven applications is that they are frequently idle waiting for user input such as mouse clicks and keyboard actions. We would like to use this idle time to preempt future user actions and create the remaining views. However, we want to also ensure that any user input is responded to immediately. The X windows libraries allow *work procedures* to be queued so that they are invoked during idle periods. However, they are not preempted as threads are, but must voluntarily relinquish control to the event loop so that user actions can be processed. As a result, they must either arrange for themselves to be called repeatedly during idle times and so save state in global variables across such invocations or alternatively risk blocking the handling of user actions while they are in progress. These work procedures are queued and one must run to completion before another can be invoked.

A threaded approach to this situation is significantly simpler. The initial view is created at startup time and one or more threads are started to create each of the other views depending on whether they are to be created concurrently or sequentially. The event loop is run in the main thread which has a higher priority than any of the subsidiary threads creating the secondary views. This ensures that a the event loop is scheduled almost immediately after an event is queued, preempting any of these other threads. So there is little potential for the event loop to be blocked from handling user actions. Also, since the tasks to create threads are run in their own thread to completion, they do not have to store their state between invocations. Thus the style of programming is more natural and the number of global variables reduced. Finally, these threads can run concurrently so that the developer need not assume an order in which the views are most likely to be needed by the user. Instead, the views can be created with equal or variable priority.

In the introductory statistics education project mentioned above, one module involves creating several views arranged as a stack of cards in which only one is visible at any time. The first card is created immediately and used to display text describing the contents of the module to the student. The remaining cards in which the user does the work described in the text are created asynchronously. The intention is to create all of these by the time the user has finished reading the contents of the first splash screen. Using this approach, the user's perceived startup time is reduced from 16 to 2 seconds[3] and perhaps by an even large factor when the application is run across the web as a Java applet.

Another situation with which most general users are now familiar and which will likely become more common in a statistical context is the concept of remote data. The recent surge in the popularity of the use of the internet allows data to be stored on a server and accessed from a remote client. Storing the data on and accessing it from a central server ensures that all users are working with the same up-to-date version. Also, in some circumstances, the data is held securely

[3]On a Pentium based PC running the Linux and the port of Sun's JDK1.0.2.

on the server and must not be stored on a client's disk.

As most users are aware, the popularity of the "web" has increased the volume of data available and also the average time taken to download a given object. As a result, most web browsers implement a caching mechanism in which recently downloaded data objects are stored locally. This avoids the overhead of downloading an object multiple times as the user returns to it on different occasions in the session. In a similar manner, remotely accessible data objects could be accessed indirectly through an extended S database designed to deal with remote objects. If the object did not exist in the database, it would be created in some suitable manner. This can involve connecting to a remote machine and using some protocol to download bytes from which the S object is created. If the object does exist locally, the database must ensure that it is consistent with the version from whence it was downloaded. Similarly, local modifications can be handled in different ways, updating the remote source copy or not.

While these two situations may seem very different, involving local computations and retrieval of remote data, they share one important characteristic. In both cases, the object being created must be stored somewhere available for future retrieval. For this, we use an S database and assign the object using some name agreed upon by all tasks that employ it. Secondly, if one task attempts to get a particular object from the database which has not yet been downloaded, the database must start the download and mark the object as pending. This allows other tasks to make concurrent requests for the object without causing multiple downloads of the same object. Finally, the object being requested can be created or downloaded in a separate thread created by the database rather than by the task requesting the object.

We turn to how this might be implemented in S. We create a new class `ObjectManager`. An instance of this class — a manager — behaves in much the same way as the evaluator manager. The constructor function for an `ObjectManager` object might be implemented as follows.

```
ObjectManager = function() {
  manager = new("ObjectManager")
  manager@threadLock = threadLock()
  manager@thread = thread(action=processManagerRequest)
  e = Quote({})
   e[[1]] = substitute(attach(d,pos=1), d=manager@threadLock)
   e[[2]] = Quote(assign(".Pending",named())
   e[[3]] = substitute(assign(".Manager",m),list(m=manager))
  sendTask(e, manager@thread)
 return(manager)
}
```

First, a `threadLock` object is created in which the objects being managed will be assigned. Also, an empty named list of the objects currently being created in separate worker threads is assigned to this database using the name `.Pending` in an attempt not to conflict with other objects stored in this database.

Note that the thread has no S-level loop to process the requests sent to it by clients. This is done by the standard C-level task loop that extracts the tasks from the queue. However, these are processed differently based on the `action` argument to `thread()`. The function `processManagerRequest()` described below is used to evaluate these requests. A point of marginal interest is how the three tasks are merged into one object (`e` in the example above).

Objects are retrieved from the manager by a client thread using the `getFromManager()` function listed below. The idea is simple. A special task is sent to the manager thread which is in the form of the object being requested supplied to the `getFromManager()` via the `name` argument. This is a self describing object in that the name used in an assignment can be retrieved from the object and also it can be passed as an argument to the generic function `create()` in order to create its contents. Such objects must be of a class that extends the class `Manageable`. For example, an object of class `htmlDocument` might specify a uniform resource locator (URL) as one of its slots and the `create()` function would connect to the appropriate server specified in the URL to retrieve the document. The task is sent to the manager thread and the client waits for it to be serviced. If the returned object is not the object `NULL`, it is the completed object. Otherwise, the object is being created in a separate thread. In this case, the client thread waits for the assignment of the object to the manager's `threadLock` by the worker thread creating the object.

The signature of this function is the same as `get()` so that we can use this as a method for that generic.

```
getFromManager = function(name, where = NULL, frame, inherit = F,
                           immediate = T, meta = 0, mode = "any")
{
 val = sendTask(name,thread=where@thread, wait=T)
  # check if object is null which means that it is being created.
 if(is.null(val)) {
  getLock(manager@lock, Name(object))
    val = get(Name(object), where=manager@lock)
  yieldLock(manager@lock)
 }
 return(val)
}

setMethod("get",signature(where="ObjectManager"),getFromManager)
```

The methods for the generic functions `exists()` and `assign()` are relatively straightforward to implement. The semantics of these functions however are dependent on the context of the data being managed by the ObjectManager object. For example, a call to exists() might wish to query whether the object has been either downloaded or pending or alternatively only the former. Similarly, certain types of objects might be considered read-only so that assignments should not be propagated to the source of the object. Instead, the assignments would be stored locally and perhaps even apply only to the thread. Accordingly, the methods for these generic functions need to be constructed with respect to the class of manager with which they will be used. Irrespective of their semantics, they can be implemented directly by accessing the manager's threadLock object or by sending the appropriate call object as a task to the manager's thread. We should note that if the manager's thread is the only thread to actually manipulate the database, then no locking is necessary. However, this is difficult to arrange in the get() method above.

The function that processes each message sent to the manager, specified as the thread's action function (processManagerRequest() by default), is given below. Messages can be in the form of objects of class call or objects that extend Manageable. If the message object is of class call, it is evaluated in the standard manner allowing calls to exists() and assign() to be handled transparently using the manager's database as the target. If a Manageable object is supplied, the manager determines if the object is already present in the database. If so, it returns it. Otherwise, it checks to see if the object is currently being created by another thread by checking the .Pending object. If this is not the case, a worker thread is established and the object is created by a suitable method of the generic create() in this thread. That thread also assigns the object to the manager's database rather than inventing another message. In a more complete implementation, we would create a more elaborate message scheme with different classes for the different message types.

```
processManagerRequest = function(obj) {
   if(is(obj,"Manageable")) {
     getLock(1)
        nm = Name(obj)
        if(!exists(nm, where=1)) {
          # check if it is pending
         if(is.na(match(nm, ".Pending"))) {
          tmp = thread(substitute(
                         managerAssign(name,create(obj), manager),
                         list(name=nm,manager=.Manager)))
           Threads = add(Threads, tmp)
           .Pending[[nm]] = tmp
         }
         val = NULL
        } else
```

```
            val = get(nm,where=1)
      yieldLock(1)
    } else if(is(obj,"call")) {
      val = standardEval(obj)
    } else
       val = NULL

 return(val)
}
```

The function that assigns the newly created object to the `ObjectManager`'s database is relatively simple. It acquires the lock on the database, and makes the assignment. Finally, it removes the element in the `.Pending` list.

```
managerAssign = function(name, obj, manager) {
 getLock(manager@lock)
    assign(name,obj, where=manager@lock)
     tmp = get(".Pending",where=manager@lock)
     tmp = tmp[-match(Name(obj),names(tmp))]
    assign(".Pending", tmp, where=manager@lock)
 yieldLock(manager@lock)
 return(obj)
}
```

As with all S databases, regular `threadLock` objects can be replaced with an appropriate `readerWriterLock` object introduced in chapter 3 section 3.2.3.

4.3 Automated Synchronization

We have seen several examples in this work and can cite many others that illustrate the difficulties associated with developing multi-threaded applications. While our feeling is that experience does make it much simpler than it might initially seem, we of course strive to make it simpler. That is the purpose of the work presented in this thesis. However, further simplification can be achieved by using the S functionality to automatically identify the expressions in a user's code that require synchronization and which can be parallelized. S is an ideal language for this since every object is an instance of an S level class. This allows each expression object to be decomposed into its sub-objects and to identify the semantics of the expression. We can use this in much the same way as certain compilers can parallelize certain kinds of loops.

Consider that we have k S expressions or tasks, $\mathtt{e}_1, \ldots, \mathtt{e}_k$. We analyze each of these expressions and determine two groups of S objects — the inputs and outputs. The set of input objects are those objects that are passed as arguments to the task or explicitly referenced within the task as global objects (via a call to `get()` or implicitly found in a database). The set of output objects consists of any assignments within the toplevel task expression and any objects explicitly committed within the task via calls to `assign()`. If we knew these two sets for each of the k tasks, we could then ensure that each reference to objects within these groups in each task is evaluated within a critical section protected by a suitable `threadLock` object. Similarly, we could determine which tasks should be serialized due to their extensive reliance on shared objects and which tasks need only be synchronized due to their relatively small number of shared objects.

In order to perform this partially automated synchronization, we must be able to determine these two groups of objects for the different code blocks. Due to the structure of the S language and the objects it creates, this information is obtainable with a small bit of work. We examine how this might be done in the next few paragraphs.

Consider the following S code:

```
10      for(i in 1:10) {
          z[i] = foo(x[i], y[i])
        }
```

This is an S object of class `for` and has length 3. The third slot or element of this object is the body of the loop and is an object of class "{". This is the generic expression block class which contains any number of arbitrary sub-expressions. The i^{th} sub-expression is accessed as the i^{th} element of the list — `x[[i]]`, where `x` is the object of class "}". In this case, there is only one expression in the body and it is of class "`=`" indicating an assignment. Similarly, the expression

```
assign("b",b,where=database)
```

is an object of class `call` and its first element is of class `name`. The left hand side of the assignment expression in the loop above is itself an object of class `call`. The function involved is the subset operator `[()`. This allows us to identify which component of the object `z` is being assigned the value on the right hand side of the expression.

We need not stop at the toplevel expressions. Since `foo()` is an S function, it is an S object. It also has its own representation and its body can be accessed via the S expression

```
foo[[length(foo)]]
```

The result of this expression is an instance of the S class "{". Thus, this can be analyzed in the manner above and so on recursively down through all function calls. Of course, this does become more complicated in view of the different evaluation frames and the lazy evaluation. However, these details can be developed with suitable attention and we need only examine these internal functions for references to non-local objects in order to determine the elements of the input and output object groups for the overall task.

Now consider two other tasks e_2 and e_3 given by the following expressions, respectively:[4]

```
{ x = scan("Xfile");  y = scan("Yfile") }
```

and

```
MyPlot(z, lambda)
```

Clearly, the outputs of e_2 are `x` and `y`. Similarly, the inputs to e_3 are `z` and `lambda`. At this point, we can automate the editing of the code to create the necessary synchronization calls. We create `threadLock` databases containing the relevant objects from these input and output groups for the three tasks. Also we add calls to `getLock()` and `yieldLock()` at the appropriate point. This facility for modifying the expressions is an available feature of S.

Graphical User Interfaces

One of the motivations for this automatic synchronization mechanism arises from GUIs. An equivalent view of these applications to that displayed in figure 4.1 is as a tree of expressions to be evaluated in response to the different user actions. The tree is characterized by nodes associated with different components that make up the physical interface. These are buttons, lists, sliders, etc. and container widgets. Each node consists of a list of lists. The elements of the top-level lists denote the type of user action such as mouse click, key-press, scrollbar motion, etc. Each of these elements is itself a list of S expressions to be evaluated on receipt of the corresponding user action. These expressions can be evaluated in multiple threads as we have illustrated above (see 4.2.1). However, the developer must be careful in associating the different expressions with different threads to ensure the necessary synchronization between the tasks. The above approach of determining the input and output object groups for each task allows this to partially automated, and the tasks to be grouped according to their use of common objects. Each group would be associated with a single thread and tasks within a group would be dispatched to that same thread.

A developer can add new interface components, such as menu items or buttons, to add functionality to the GUI at any stage in the development or even to another application. However, there is always a danger that a task added in this way will destroy the synchronization of the existing application. Having an automated mechanism to determine the appropriate thread to which the task should be dispatched eases the burden on the developer and reduces the potential for errors or a lack of synchronization.

[4]The braces { and } combine several expressions into a single expression object and are required to combine multiple sub-expressions into a single task.

Compilers and Extensions

Many of these concepts have been present in compilers which are capable of parallelizing loops and other constructs in the target language(s). However, the output from these compilers is less amenable to the user since it is simply object code. The types of tools we refer to above can provide output to the user from each of the intermediate steps and can be easily integrated with other tools due to the high level nature of the S language. Additionally, this set of tools has the potential to be more general than compiler-based parallelization as more information can potentially be retrieved from S about an expression due to its functional nature. This feature also makes this overall approach more complex since S is dynamic and functions and other objects can be modified within an evaluation rather than being statically linked before execution. We must constrain the situations in which this semantic analysis is applied to situations in which a certain evaluator configuration (e.g. search path, basic functions, etc.) is constant.

The ideas described in this section can be implemented and extended as a relatively complete suite of tools for analyzing and characterizing S expressions. By being able to identify expressions that must be serialized or synchronized, we can make the development of multi-threaded applications simpler and more accessible to casual users. Additionally, the tools can be used by individuals to prototype algorithms using S that will eventually be rewritten in a lower level language, such as C. Since these low level languages do not typically provide user support for threaded application development in any form other than the API, such a suite of tools could be very useful, if adequately developed.

Chapter 5

Internal Implementation

This chapter provides details about the low-level implementation of the thread facilities — classes and methods introduced in the previous chapters. We start by examining the structure and behavior of a single S thread and then focus on the mechanism by which threads communicate. This mechanism is embodied in the *evaluator manager* which is also used to control and govern the threads in the S environment. Before turning our attention to the `threadLock` implementation, we discuss the modifications necessary to allow the implementation work in the multi-processor (parallel) and multi-machine (distributed) computing environments. We offer advice for S programmers developing C/C++ based routines so that they can be used in both the threaded and non-threaded S environments. Finally, we discuss some miscellaneous changes and the necessary system requirements to run the threaded version of S.

5.1 Internals of a Thread

We have seen that the S function `thread()` is used to create an S level thread. This in turn creates an S evaluator and a C level thread in which the evaluator operates. In this section, we look at the structure of this C level task.

The C structure `EvalThread` is used to encapsulate all the information related to an S thread. This is the top-level structure that contains both the evaluator used to process each task and the "between tasks" data. These include the task queue in which tasks are sent from other threads, the task history list and the parent and children thread information. The task queue is a simple form of a linked list with each element of type `SThreadTask`. The S evaluator contains fields related to the evaluation of a single task and thread-specific "environment" variables. These include the call stack frames, local method tables, standard connections, search path and the thread status.

A call to `thread()` maps to the creation and appropriate initialization of an `EvalThread` object. When the C level thread is created, it starts executing a routine. This routine is called with the `EvalThread` object as its only argument. The body of the routine is quite simple. It consists of an infinite loop. Each iteration retrieves the next element from the task queue, waiting until such an element is present. The task is then evaluated and the task and result added to the history list. If the completion of the task is being waited on by another thread, the task's source thread — the thread that sent the task — is signaled.

When the thread is created, the task queue is empty. The thread is therefore IDLE and waits for an element to be inserted on the queue. The fact that the body of the routine is a loop allows us to associate the C level thread with more than one S task. Since the same evaluator is used in each task within this thread, we have the persistence across tasks that is useful in communication and synchronization across threads as described in chapter 3.

```
void *
S_evaluator_thread(void *data)
{
 EvalThread *thread = (EvalThread *)data;
 s_evaluator *S_evaluator = thread->evaluator;
 SThreadTask *task;
 int status;
 s_object *task_expression, *value;

    /* local initialization that must be performed within the thread */
 set_evaluator_specific_thread_data(thread, NULL);
 S_evaluator = S_evaluator = thread->evaluator;

    /* Task processing loop */
 while(TRUE) {
       /* Get the next task from the queue, waiting until
          there is one to be retrieved.
        */
    task = get_next_task(thread);

     /* Check to see if the manager has ordered this thread
        to do something (e.g. suspend or terminate itself)
        and wait there if necessary. This gives the evaluator manager
        the ability to suspend this thread.
      */
    check_status(thread->evaluator);
```

```
        if(thread->evaluator->status == Status_TERMINATE)
            thread_terminate(NULL);


          /* Evaluate the task in the traditional S manner */
        status = do_task(task->expression, S_evaluator);

          /* Add the task and return value to the history list */
        add_task_to_history(thread, task->expression, last_task_value);



          /* Now signal the waiting thread if there is one. */
        if(task->cond_var) {
             /* Put the return value back into the message by having the
                evaluator manager copy the object into the waiting thread's
                storage area. If the thread was canceled, indicate this.
              */
           task->return_value = request_manager_copy(last_task_value,
                                                      task->source_thread,
                                                      CURRENT_EVALUATOR_FRAME,
                                                      S_evaluator);

              /* Announce that the task has finished. The other thread
                 waits on this condition. */
           task->answered = TRUE;

           COND_SIGNAL(task->cond_var);
        }
     } /* End of the task loop */

    return(NULL);    /* Should never get here */
    }
```

We should note that the `do_task()` implicitly evaluates the task using either the standard evaluator or the S function passed to `thread()` via the `action` argument (see page 25). This is done in the S evaluator and the `Eval()` routine irrespective of the threading. Other than this we will not discuss the specifics of the evaluation of the task as it is documented in [4] and the modifications mentioned in chapter 2.

Within each iteration of the task loop, after returning from retrieving the next task, the status of the thread is checked. The evaluator manager can set this status indicator to signal the thread to behave in some particular manner, such as SUSPEND, etc. (The evaluator manager will typically do this on behalf of another thread using a call to `threadStatus()` or for its own house-keeping purposes which may require the thread to be in a particular state.) At this point, the thread may block waiting for the evaluator manager to modify the status to a state in which it can continue. Alternatively, the evaluator manager may terminate the task or thread. In the latter case, the routine will exit killing the C level thread. Generally, the status of thread will be "CONTINUE" and the thread will proceed to evaluate the task.

In the above, a thread only checks its status between processing tasks. This would not provide the necessary control to suspend or cancel a task while it is in progress. In order to do this, we have modified the `Eval()` routine to check the thread's status each time it is called. Since this routine is called recursively within a task, this provides more than adequate resolution to control the evaluation of task. The same semantics apply here as in the check implemented in the task loop above.

The history list contains elements consisting of a task and the return values of each that task evaluated by a thread. Each element also includes a task identifier which is a sequential index in the task queue. This is used to compare tasks in this list with `threadTask` objects supplied as arguments to S function calls. This list is list with length specified by the thread attribute `maxHistoryLength`. As elements are added and the list's capacity exceeded the earliest elements are removed, conserving the number of elements[1]. The elements in the list are used by the `history()` function and also in calls to `join()`. In a call to `join()`, the calling thread first compares the target task with each of the tasks in this list belonging to the target thread. If a match is found, the return value associated with the matching task is returned. In the event of no match, the calling thread conditionally waits for notifications of the target thread completing a task and matches that task.

In addition to this history list of previously evaluated tasks, the thread makes the current task object available to the evaluator manager. This allows other threads to query the value of this object using the function `currentTask()`.

When the source thread of a task is waiting for a response, the evaluator thread appears to communicate directly with this waiting thread. However, we consider this to be an indirect form of communication. The evaluating thread does not have access to any of the other thread's internal fields and the communication is done using the structure of the shared task object. Indeed, the evaluating thread need not even be aware of the fact that it is another S thread waiting for the signal it dispatches. Instead, all communication between two S threads is done indirectly through a central mechanism called the *evaluator manager*. We describe this in the next section.

[1]The value of `maxHistoryLength` can be set dynamically using `setThreadAttributes()`

5.2 The Evaluator Manager

The evaluator manager controls the threaded S environment by managing each of the threads and performing operations on different threads on behalf of another thread. It also keeps or can compute the current state of the S environment and can be used to monitor the state of the different evaluations and overall process. The evaluator manager consists of a separate thread whose sole purpose is to processes requests or messages from each of the S (evaluator) threads. In this sense it is an information server and broker for inter-thread communication. Among the types of information available from the manager are the existing threads and their statuses, the current owner of one or more `threadLock` databases, and potentially which lock a particular thread is waiting to acquire. The manager also performs the operations which implement inter-thread functions such as `sendTask()`, `join()`, `threadStatus()`, etc. and the query functions such as the history

As the manager is processing one request from its message queue, all other unrelated threads waiting on a response to other messages in the queue are also blocked. This is an inefficiency that can be avoided by allowing direct communication between threads. This is the canonical alternative approach. If one thread wishes to query or set a field in another thread, it must send a request to the manager which relays the query or operation. A direct channel between the two threads would avoid the fractional overhead of involving the manager. More importantly it would remove the global queuing of requests and make it local to the target thread decreasing (or at least not increasing) the response time. However, we believe that the more structured client-server model is beneficial in the context of a flexible language such as S that can be used in arbitrary ways. Without it, diagnosing errors or anomalies becomes significantly more challenging. Additionally, for the reasons mentioned in the previous paragraphs, the current design works well when one considers characteristics other than efficiency such as extensibility and maintainability. Other advantages are discussed below.

The evaluator manager behaves analogously to an evaluator thread in that it processes elements of its message queue inside an infinite loop. The queue is a linked list of `ManagerMessage` objects and this structure and its contents are described in the next section. For the moment, we need only think of these objects as containing information about what operation to perform and on what thread(s). A critical aspect of the evaluator manager is that it is intended to process a large number of short or quick requests. In this way, it is always available to process more requests and any thread waiting for a response from the request will be answered rapidly, allowing it to proceed without an excessive period of inactivity.

There are several reasons why we use the evaluator manager rather than other more efficient or obvious approaches. The most important advantage of the evaluator manager approach is simplicity and caution. The implementation described above provides serialized access to the internal structures controlling and describing the S environment. This is because the requests or messages are processed sequentially by just one thread. Allowing just one thread to maintain these structures removes the need to provide a synchronization mechanism to limit access to them. Instead, the synchronization is moved to a higher level which involves having a requesting thread wait for the response to its manager request. Overall, this synchronization is significantly simpler to manage and debug and involves approximately the same number of locks and overhead as other approaches.

A second reason why the single manager approach is useful relates to how we query state information. The evaluator manager maintains a table mapping thread identifiers used at the S level to `EvalThread` objects used at the C level. The direct communication mechanism would either require this table to be maintained using reader-writer locks to ensure synchronization between the different threads or alternatively force the `EvalThread` objects to be embedded into the S objects that represent threads. While the latter is possible (using `raw` objects introduced in S version 4 [12]), it entails validating pointers potentially from different sessions and maintaining another global table. Additionally, if the state information is not stored in a central area querying it necessarily involves polling each thread for the information. This is inefficient and complicates the thread's task loop described earlier.

We mentioned briefly in a previous chapter that the evaluator manager would cache redundant threads and reuse them in subsequent calls to `thread()` rather than terminating threads. This avoids the overhead of creating both a C level thread and an S evaluator. In the absence of the evaluator manager this would not be possible, or would be replaced by a version of the manager that was only concerned with managing the creation and destruction of threads. In either case, the added complexity or the lack of reusable threads would be an unfortunate feature.

The evaluator manager allows us to think of one thread being in charge of many others in the traditional master-slave model. It is possible to extend this model so that we can have a collection of evaluator managers themselves managed by a single evaluator manager. Each of these sub-evaluator managers would manage its own group of threads. This structure could be used to provide a form of protection and containment of threads within an application in the larger S environment. Additionally, such a structure is natural when we consider distributed computing. Each machine in the array of machines would have its own S process and unique evaluator manager. A single manager would be responsible for controlling each of these slave evaluator managers. Each of these slaves however would manage its own collection of threads. The key point here is that the model is scalable and allows replication. We return to this below.

While not a unique feature of the evaluator manager, an important aspect of the evaluator manager is its extensibility. The types of messages it can handle can be easily augmented and the character of the manager modified. We explain this message handling procedure in the next section.

5.3 Messages and Methods

In this section we turn our attention to the details of the messages processed by the evaluator manager. Each thread has the identifier of its evaluator manager. It uses this to insert a request on to the manager's message queue using the routines `QueueThreadRequest()` and `PutNextRequest()`. [2] The messages are either requests or commands. Each message is concerned with information about either the state of the environment or the contents of one or more threads. In the former, the manager can respond to with local information and in the latter case, the manager must compute the information from the thread structure to which it always has access. In some cases, it must suspend the target thread before it can compute the necessary information.

The mechanism by which the appropriate handler or routine is invoked to handle a message is a poor man's version of object oriented programming in C. We create an array of routines indexed by the message type. We index the relevant routine after we have determined the message type and invoke that. The signature for the handlers is by necessity the same since we are using C rather than methods in C++[3]. This signature is

```
boolean (*handler)(ManagerMessage *msg,
                    EvaluatorManager *manager, s_evaluator *S_evaluator)}
```

Thus the message, manager and the source thread's evaluator are provided to the message handler.

A message is a C object of type `ManagerMessage`. This has the following structure.

```
typedef struct _ManagerMessage {
 /* Indicates that the message has been processed */
  boolean answered;
 /* The type of message - see  ManagerMessageTypes in the table below  */
  ManagerMessageTypes type;
 /* Any data provided by the requesting thread for the operation */
  void *data;
 /* where to put the return value. If this is NULL, the requesting thread
    is not waiting and no return value is required.  */
  void *return_value;
 /* Whether the requesting thread should free the return value. The  manager
    may allocate the return value's storage space in the source thread's
    storage area - either the current frame or permanent storage frame. */
```

[2]It does this by acquiring an exclusive lock on the queue and appending the message to its last element. Then it signals the a condition variable on which the manager is waiting so that the latter is aware that there are elements in the queue.

[3]We chose not to use C++ in order to preserve the simplicity of the installation process. Additionally, the so-called Standard Template Library is not as well supported by the GNU compiler **g++** as we would like. We decided that simplicity was the better side of valor and not to confuse the introduction of threads with more reorganization of the source code.

```
  boolean you_free_it;
 /* The thread that made the request and put this on the queue */
  EvalThread *source;
 /* Necessary facilities for locking and notifying the queue listeners */
  MUTEX *lock;
  COND_VAR  *lock_var;
 /* Next element in the linked list or queue */
  struct _ManagerMessage *next;
} ManagerMessage, *ManagerMessageQueue;
```

Each message identifies the source thread from which it came. Arbitrary objects can be used as the message-specific data and it is passed directly to the message handler routine without modification. The only necessity is that the creator of the data and the handler agree on its type. The `return_value` is used to store the object returned from the request and again the type of object used here must be agreed upon by the producer and consumer.

If the source thread is waiting for a response rather than issuing a request for an asynchronous action, the routine `new_manager_message()` initializes the variables `lock` and `lock_var`. These are used to synchronize the manager's activity with respect to this message and the source thread. A thread must wait if a response is expected or, for example, if the data object must be copied into the target thread's storage area. The latter is necessary in a call to `sendTask()`. Figure 5.1 displays this synchronization mechanism. A request from thread A to the manager uses the manager's queue locking mechanism to append the message to the queue and signal the manager. The thread awaits a response by conditional blocking on the value of the message's `answered` field. It does this using the following loop.

```
   /* If we are waiting for a response */
 if(msg->lock) {
     /* Grab the lock and test to see if the answer has been processed */
   LOCK(msg->lock);
   while(msg->answered != TRUE) {
       /* Release the lock and wait for a signal. When this call
          returns, the lock will be owned by this thread. */
     COND_WAIT(msg->lock_var, msg->lock);
   }
   UNLOCK(msg->lock);  /* Release the lock, although it doesn't matter */
    /* Destroy the message */
   destroy_message(msg);
 }
```

Thus the manager communicates with the waiting thread using the lock contained in the message.

The situation in the call evaluated within thread A

```
sendTask(expression, thread=B, wait = T)
```

illustrates a slightly more complicated version of the communication also shown in figure 5.1. The message is communicated to the manager in the same way as above. However, the response does not come from the manager but instead the manager's handler for this type of message forwards the request to the target thread, B in the form of an entry on its task queue. A field in the task structure contains the lock, condition variable and `answered` variable on which thread A is waiting. Thread B then uses these variables to signal thread A when the task has been completed. Before thread B notifies thread A, it may wait for the evaluator manager to copy the return value into A's storage area. This mechanism does not contradict the claim that no two threads can communicate directly as thread B is unaware of what thread is listening for the completion signal and is incapable of utilizing information about thread A.

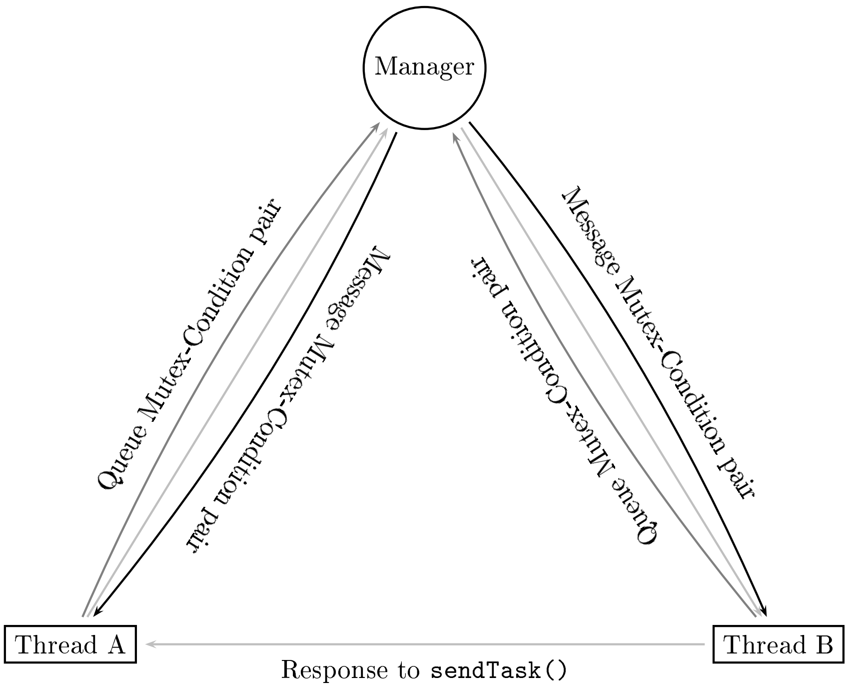

Figure 5.1: Evaluator Manager Communication

Not every message contains a target thread as some requests are to the manager and others refer to more than one target thread. Accordingly, this information is in the message specific data.

Table 5.1 gives a list of the message types and a brief explanation of their purpose[4].

Table 5.1: Manager Message Types

Type Identifier	*Description*	*Used In*
SESSION_QUIT	request that the session be terminated and other threads be notified to terminate themselves.	`q()`
SET_STATUS	set the status of one or more threads.	`threadStatus()`.
GET_STATUS	retrieve the current status of one or more threads.	`threadStatus()`.
GET_THREAD_OBJECT	retrieve one of several objects associated with a thread. These include the task history list, current task, last task result, `.Last.value`, search path and task queue.	`currentTask()`, `taskHistory()`, `taskQueue()`, `lastValue()`, `valueOf()`, `searchPath()` and the vector of object type identifiers `ThreadObjectValues`.
REMOVE_TASK	remove a task from a thread's task queue.	`removeTask()`.
SEND_TASK	insert a task on a thread's task queue for future evaluation.	`sendTask()`.
COPY_OBJECT	used internally to copy an object from one thread into another's storage area at the end of a call to `sendTask()`.	`sendTask()`.
CHILDREN	return the children threads of a given thread.	`children()`.
PARENT	return the parent thread of a given thread.	`parent()`.
HAS_ACCESS	determine whether a thread has access to a set of other threads.	`access()`.

[4]A complete description of the messages and their associated data types will be provided after we have extended the evaluator manager design to work in a distributed computing environment if this information proves to be useful to an advanced user.

Type Identifier	*Description*	*Used In*
SEVAL	have the evaluator manager evaluate an S expression in the manager's own evaluator. This is used to initialize the S process and also to modify the state of the manager for objects stored as S objects.	not used directly at this point except for initialization.
SET_ATTRIBUTES	set a thread's attributes.	`setThreadAttributes()`.
GET_ATTRIBUTES	retrieve the current values of a thread's attributes.	`getThreadAttributes()`.
JOIN	wait for a set of tasks to be completed.	`join()`.
NEW_THREAD	create a new thread (or recycle an old one).	`thread()`.
MEM_EXCEEDED	inform the manager that the source thread has noticed that the memory has exceeded the limit for the process and that it should restart itself by dumping all the threads for a reload.	internal message in `Eval()`.
THREAD_LIST	return a list of all the threads in existence in the environment.	`threads()`.
LOCKS	retrieve a list of all the locks and owners of same.	`LockOwners()`.
LOCK_OWNER	retrieve the current owner of a given `threadLock`.	`lockOwner()`.
REGISTER_METHOD	add a handler method for a message type to override or extend the manager.	done in user's C code at present.

Other methods are present to handle `threadLock` actions which can be used to centralize this mechanism in the evaluator manager. This will probably be a necessary change to support distributed computing (see section 5.4 below).

One of the basic philosophies of the messaging system is that the handler invoked within the evaluator manager's thread should do the minimum amount of work to process the request. This basically involves performing as much of the pre-processing of the message in the calling/source thread. This can be done in the creation of the message at the C level or in the S functions or a hybrid approach using both arenas. For example, thread objects should be coerced to the native type used by the evaluator manager before being inserted into the message as part of the data. This is done in the S function `threadIds()`. Also, if the length of the returned object is known by the source thread before the message is processed, the `return_value` field should point to allocated memory to hold these objects or values. The S functions that communicate with the evaluator manager all use the `.Call()` ([12]) interface to invoke the appropriate C routine and manipulate the associated S objects directly.

The choice as to whether the message request should be vectorized or not in the target threads argument is not an obvious one. One the one hand, the processing time for this one message will increase and so too the latency in the manager responding to other messages. However, overall the number of messages will decrease. The philosophy we used was to vectorize messages for which the response was a simple object. For example, the message type `PARENT` returns a vector of threads identifiers (basically integers) and can be simply vectorized by allocating an equal number of objects to `return_value` as there are target threads. However, the `CHILDREN` message returns a ragged array of thread identifiers and so was left as a non-vectorized message capable of handling just one target thread in a single message.

5.4 Different Models — Single, Multiple and Distributed processors

The initial implementation of threads in S was done using a user level threads library. This means that the threads are not managed or scheduled by the kernel but instead compete for the resources only within the process in which they are created. The library includes a scheduling algorithm and a mechanism by which threads can be preempted and reinstated onto a processor. Kernel level threads, as the name suggests, are owned and managed by the kernel and compete for the system resources with the other threads in the entire system. While they are also created within a process, they are visible to the kernel. Since user level threads are simply structures within a single process, it should be clear that they share the same memory and symbol space. The kernel must implement this for kernel level threads and allow the threads to share the same memory.

Since the kernel cannot be aware of user level threads, it cannot schedule them or associate them with a CPU. Accordingly, user level threads do not permit explicit parallelism, but only concurrency. While they may allow a performance improvement as mentioned in chapter 1, the computational applications mentioned in chapter 4 require the ability to use multiple processors to realize any significant increase in performance. This requires kernel level threads, each of which can be be bound to an available CPU.

An early approach which allowed us to proceed in developing the user level thread framework described above was to:

- put the global, thread-specific global variables into a a global structure and change the references in the code to expand (via the pre-processor) to dereference the relevant field in the global variable
- change the contents of the global evaluator variable at each context switch to reflect the newly active evaluator

Having identified the variables that were thread specific, the first step was relatively simple.[5] The second stage involved overriding the context switch in the user level library routine. While this was relatively straightforward since the library (Pthreads by [36]) was suitably structured, it requires that we track updates to libraries other than our own. It also committed us to this library or to support modifications to numerous user level thread libraries, such as DCEthreads, etc.

A second stage in the development allowed us to use kernel level threads. The task involved removing all references to the global variable `S_evaluator` which pointed to the currently active thread's evaluator. Clearly, with only one such variable, only one evaluator could be active at a given time. Instead, what was needed was to make the evaluator available as a local argument or object in each of the C routines that required it. Of course, if routine `A()` calls `B()` which in turns call `C()` and `C()` requires a reference to the evaluator, that reference must be passed through `B()`. After extensive modifications to the the signatures of a large number of routines in the S source code that removed the reliance on thread-specific global variables, the code can now be linked to use kernel level threads.[6] These changes can potentially affect users' native code that calls routines that require access to the thread's evaluator. We address this problem in section 5.6.

An other approach to implementing S level threads is to create our own scheduler. This would be useful in order to increase the portability to other operating systems such as Windows NT and Win95. However, implementing our own scheduler either causes difficulties with recursive calls to `Eval()` and increasing stack depth or requires that we implement the same sort of stack re-instatement as in user level thread libraries. It is not clear how deal with the former and the latter gives rise to the same portability problems we are trying to avoid. Also, the Pthreads standard is exactly that — a standard — and so is likely to be implemented on several platforms. Currently, implementations exist for many of the hardware systems commonly used to run UNIX/Linux. Additionally, we have written the code with an eye on allowing easy interchange of thread programming interfaces as described in section 5.6.

[5] This simplification of the process was the idea of John Chambers. He prepended an underscore (_) to the name of each field in the global structure `s_evaluator` to avoid any spurious matches in the code with the existing names.

[6] Using the LinuxThreads package which implements kernel threads in Linux, this did not behave correctly when we first tested it and we have not returned to explore the difficulties. However, conceptually all the necessary modifications are in place and the cause of the problems most likely is attributable to external libraries in the system.

Distributed Computing

The ability to utilize multiple CPUs has the potential to make a significant contribution to some areas of statistical computing. However, it does require suitable hardware that provides access to multiple processors within the same machine. These are expensive and accordingly more rare than single processor machines. While the availability of multi-processor machines is increasing (especially given the Intel MP spec and relatively cheap motherboards using Pentium processors), the most common environment that users have access to is a network of workstations. Accordingly, we would like to extend the concepts presented in this thesis to work in this distributed environment.

The basic idea in distributed computing involves multiple processes on one or more machines. The processes communicate using some protocol using some communication interface (such as shared memory, RPC or an application specific method). Tasks are sent to the different processes either from a master process or from sibling processes and results are communicated in response to these tasks.

S comes with its own protocol — the S language and S expressions — and this can be used as the communication protocol between the processes. The communication mechanism is via some form of an S connection This might be a FIFO [37] if the processes share a file system or otherwise a socket. The results of a task can be communicated using the same mechanism.

As we outlined above, a natural structure for distributed computing is to have an evaluator manager act as the master of all the processes and to have it broadcast the tasks to to the different slave processes. Each of these slaves also has an evaluator manager which communicates with the master manager and coordinates its own set of threads. To implement this, we have to extend the evaluator manager so that it can spawn new processes on other machines and create remote threads indirectly through the slave evaluator managers. Also, we must extend the `thread` and `threadTask` classes to reflect both the process and machine in addition to the thread and task identifiers.

Another difficulty we have in this context is synchronization. Since the synchronization primitives are implemented using `mutex`es and condition variables that rely on shared memory, these will not extend to a distributed system. However, if implemented through corresponding higher level mechanisms in the evaluator manager (using semaphores [19] and appropriate messages), these concepts will provide the same functionality in the distributed environment as in the threaded environment described in the previous chapters.

It should be relatively clear that the modifications necessary to support distributed computing consist of extensions to the existing design. The S classes can be extended in a natural way given the object oriented nature of S. A suitable master evaluator manager can be created by modifying the message handler routines to receive the requests from a different source — the communication mechanism. Surprises and unexpected difficulties will of course arise when we turn our attention to implement this in the near future. However, we are confident that the design presented in this work will extend to support distributed computing.

5.5 Databases & ThreadLocks

There are four primitive operations associated with S databases. These are accessed through the S functions `assign()`, `get()`, `exists()` and `remove()`. An essential characteristic of these operations is that they are *atomic* with respect to the database. By this we mean that no other thread can access the database while the operation is in progress. It is relatively straightforward to see why this is necessary. If one thread is in the process of assigning an object to the database and another thread retrieving the same object, the second thread may get either the original object or the updated object. A worse situation might occur in which a place holder is reserved for the object but no associated object is yet in the database. If the second thread retrieves this object at this point, chaos may ensue. Accordingly, these operations must be atomic to avoid race conditions.

The atomicity is relatively simple to implement. We add a `mutex` field to the generic database structure which is used to ensure exclusive access. When any of these operations start, they acquire the lock. If the lock is held by another thread, this call blocks until the lock is acquired. The details of the operation are then executed and the lock released. The implementation is complicated by the fact that it is not always clear where the operation begins and ends. Also, the top-level routine that implements an action calls other routines which may be used in other ways throughout the code. These may also try to acquire the lock and if left to do so would deadlock the thread. Accordingly, these must be modified to either move the lock acquisition to their other calling routines or to take an argument indicating whether the lock is already held by this thread, or some other appropriate mechanism. These are minor details associated with retro-fitting existing code to make it thread-safe.

An S database is implemented using a hash table structure that is used throughout the S internals. We spent some time modifying this structure and the associated routines that manipulate instances so that it was thread-safe. This approach made implementing the databases and several other features significantly simpler as it provided a second and autonomous layer to the synchronization mechanism.

Since two threads can have the same database attached in their search paths, it is essential that the S engine not discard the database (from memory) when one detaches the database. However, when the database is no longer being used by any thread, the memory must be reclaimed. Fortunately, this was done in the existing pre-threaded code using reference counts. As each thread attaches the database, the reference count is incremented. Similarly, detach actions decrement this count. When the count decreases to 0, the appropriate cleanup actions are performed.

ThreadLock Mechanism

The `threadLock` framework is not a common approach to synchronization in the thread literature, but we believe is a simpler interface to S user level synchronization. It is peculiar to a language such as S which allows persistence and multiple databases and in which names are resolved to associated objects.

We add the following fields to the generic database structure and use them to provide the ordinary locking mechanisms.

```
MUTEX *owner_lock;
    /* Count the number of times this has been locked.
       This is a recursive mutex. */
COND_VAR *owner_lock_var;
unsigned long count;
s_evaluator *owner;
```

The fact that the fields are present in instances of all database types is a result of not using an object oriented language. It does consume a small number of extra bytes - on the order of 20. However, the objects are never initialized or used except in `threadLock` databases. But their presence does allow the synchronization functions to be applied to any S database. This feature may be relevant in the future (e.g. in distributed computing).

Locking

The implementation of the simple locking mechanism acts much like a counting semaphore [25]. Rather than use the `mutex owner_lock` to directly provide exclusivity, we set the value of the `owner` variable in a database's structure to indicate which thread currently owns the rights to access the database. The `mutex` is used to ensure that only one thread has write permission to this `owner` variable. Every thread that attempts to acquire access checks the value of this variable. If it's value is `NULL`, the requesting thread sets the owner to be its evaluator and proceeds. Other threads must wait until that thread releases the ownership by resetting the variable to `NULL`. These other threads are signaled using the condition variable `owner_lock_var` that the ownership status has changed. This mechanism allows them to block waiting for such an event in order to acquire the ownership.

Given that we are simply trying to effect exclusivity, why do we use this complicated approach when a `mutex` would perform the same function? This is simple to see when one considers the situation the `threadLock` class was designed to handle. A thread, A, acquires ownership of the lock in order to modify an object using an expression that is not a simple call to `assign()`. For example,

```
n = n + 1
```

The lock is necessary to prohibit any other thread from modifying the value of `n` between the time the left hand side of the expression has been evaluated and the result assigned to this object. If we were to use a simple `mutex`, the initial acquisition of the lock would cause the subsequent implicit calls to the functions `get()` and `assign()` to deadlock. This is so because these functions need to acquire the same lock or be told that the thread already holds it.

The `owner` variable approach allows the calling thread to continue processing inside a critical section if *it* is the owner of the lock. The thread examines the value of `owner` within a critical section protected by `owner_lock`. Each time the thread continues in this way, it increments the value of `count`. This is used in much the same way as the reference counting above for databases. When the thread releases the lock it decrements the count and sets the value of `owner` to `NULL` only when the count is 0. This is a recursive lock [25].

Other approaches are possible, but this is a relatively standard mechanism for implementing recursive locks. Note that we use two `mutex` objects in the database. The `lock` object is used for the operations and the `owner_lock` object is used for critical sections concerning access to the variable `owner`.

Conditional Locking

Conditional locking involves the following steps:

- acquire ownership of database
- test condition
- return if condition is true
- otherwise relinquish ownership and wait for signal on condition variable indicating potential change to condition value and repeat.

Given the slightly more complicated method of acquiring ownership of the database, the conditional locking mechanism cannot be implemented using the simple idiom

```
LOCK(owner_lock);
while( evaluate condition == FALSE) {
 COND_WAIT(lock_var, owner_lock);
}
```

Instead, the locking call must set the value of the `owner` variable and likewise the body of the loop must release the ownership before sleeping. When the thread is signaled via the condition variable and awakes, it holds the lock `owner_lock` and must attempt to acquire the ownership of the database rather than this lock. The details of this are relatively straightforward once this procedure and the differences between it and the idiom above are identified.

The routines that implement the `assign()` and `remove()` S functions broadcast their actions using `lock_var`. As we mentioned in chapter 3, this causes all threads waiting on this database to reevaluate the associated condition and this can lead to unnecessary computationally expensive evaluations if the changes to the database do not involve objects in the condition expression. Instead, we would like to be able to have a thread signaled only when an action of interest has been completed. We do this using `threadLockEvents` and `threadLock` event queues. This is very similar to the evaluator manager mechanism with the additional steps of registration.

Figure 5.2 shows how this more refined waiting is done. Thread B registers its interest

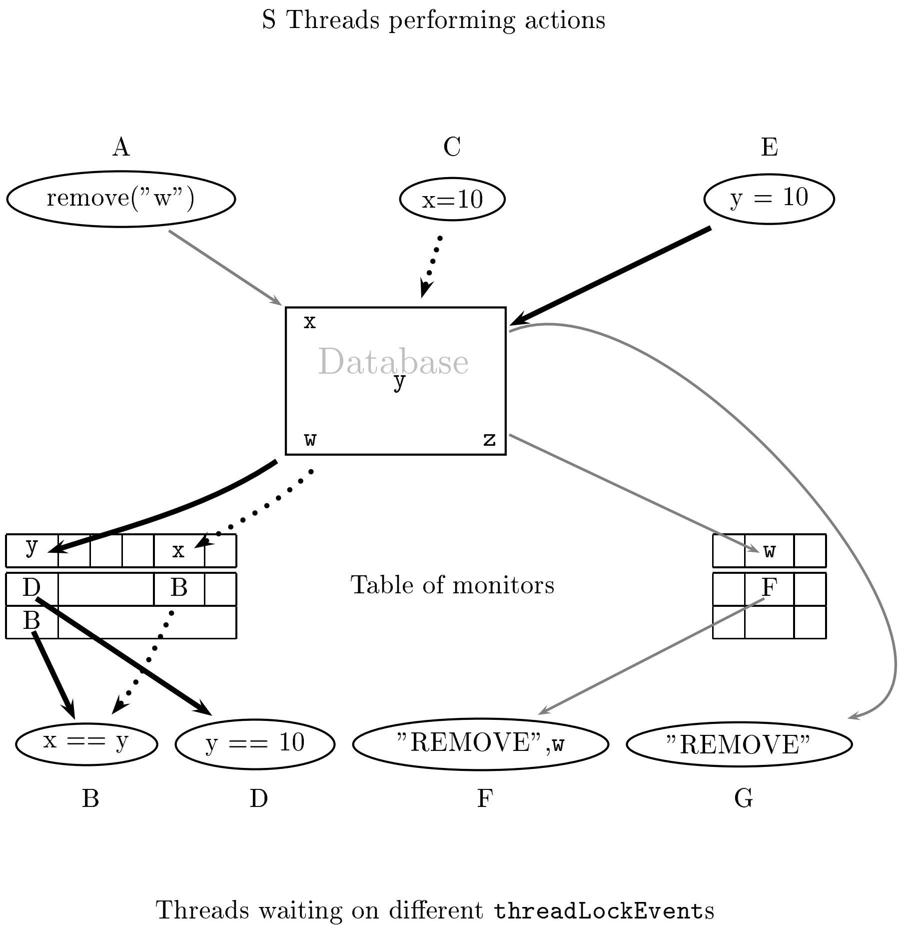

Figure 5.2: Actions and Notification via `threadLockEvent` objects

in assignments to the objects `x` and `y` with the database. At some other point, thread D registers interests in assignments to `y`, increasing the count of waiting threads on the object `y` to 2. Both threads are stored in the ASSIGN hash table indexed by object name. Thread B goes into the linked list of threads waiting on the object `y` and also the list for `x`. Similarly, thread D is an element of the list for `y`.

Thread C then assigns a new value to `x` generating an ASSIGN event for this object. The database examines the ASSIGN hash table and determines that there is an entry for `x`. It then traverses the elements in this list, notifying each element by inserting a copy of the `CthreadLockEvent` instance into the associated thread's `threadLock` event queue. At this point, each of these threads is signaled and can process the events in queue. Each waits to acquire the ownership of the database and reevaluates the relevant condition expression. If the condition is then true, the thread owns the lock and must unregister itself from the database's table of monitors.

Rather than inserting a copy of the `CthreadLockEvent` into each thread's message queue, we use reference counts within each `CthreadLockEvent` to allow the object be shared as a read only instance. The database sets the count in the event object to the number of elements in the linked list. As each thread processes the event, it decrements the count. The thread that decrements the value to 0 destroys it.

There are two `threadLock` monitor tables stored in each `threadLock` database. One is used for ASSIGN events and the other for REMOVE events. The latter works in the same way as above and notifies the threads waiting on the relevant variable using the `threadLockRemoveEvent`. The database also maintains a list of threads that are to be notified of any assignment or removal event. This is how thread G is notified in our example. These lists are not kept in the table as they cannot be associated with any object name. In the example above, thread A removes the object `w` from the database. Both threads F and G are notified, but in different ways. Thread F had registered for such events on the object and G had expressed an interest an all REMOVE events. Thus F is notified as an element in the linked list stored in the REMOVE table under `"w"`. Thread G is notified directly by the database using this list of global monitors.

This event queue/monitor table approach is very flexible in that it allows us to extend the types of events we can monitor and dispatch. Additionally, it can be trivially extended to allow each database run in its own thread. The `get()`, `assign()`, etc. operations would then also become messages or requests to the database. In processing these requests, the database would dispatch the relevant events to the waiting threads. This server model may be necessary to implement the distributed environment mentioned in section 5.4 above. It also has the advantage that registration requests can be handled in the same manner as messages to the database.

5.6 Guide to S programmers using C/C++

In general, users developing S functions or C or Fortran routines to be used in S can ignore the presence of threads. These will run within a single thread and should not need to communicate with any other thread. Any communication will usually be done in terms of other functions and routines which hide this from the developer. New threads can be created using the S function `thread()` rather than directly in the C code. If this thread is to be used to execute a C routine, that C routine can be called from an S expression sent to the newly created as a task. For example, if we want to create a new thread to evaluate the routine `foo()` which operates on a vector of integers, we might do so as an S expression in the following way.

```
thread(substitute(.C("foo", as.integer(vals), as.integer(length(vals)),
                    list(vals=input))))
```

or

```
thread(substitute(.Call("foo", vals), list(vals=input)))
```

where `foo()` is defined as

```
void foo(int *vals, int *length)
```

or

```
s_object *foo(s_object *vals)
```

respectively.

The disadvantage of creating a thread this way is the overhead involved in creating the S evaluator and processing the task request via the evaluator manager. However, the advantages include simplicity and also registration of the thread with the evaluator manager. In this way, the thread is known to the environment and can be managed, queried, etc. in the usual ways as part of the system. This approach is scalable to multiple threads used together to form one large task as a set of sub-tasks distributed across threads. It is also how we implement the event loop for a GUI, for example.

When writing C routines that employ other routines in the S source code[7], one may need to be aware of certain thread specific details. This should not be considered optional, especially if the code is to be used in both the threaded and non-threaded environments. Typically, the most immediate difficulty arises from the changes to so many of the S routines that now take the `s_evaluator` object associated with the thread as an extra argument. This problem will manifest itself as a compilation error. Also, many macros developed to facilitate development of this type of code such as `Calloc`, `Realloc`, `CLASS_AS_CHAR`, `MAKE_CLASS`, etc. require the existence of this

[7] This is only likely to be an issue for advanced developers creating routines to be used via the `.Call()` interface.

local variable, `S_evaluator`. These functions invoked through the `.Call()` or `.C()` functions must compute their thread's evaluator as a local variable. This is done using the macro `S_EVALUATOR`[8]. as a declaration at the beginning of the routine. This expands in the threaded context to

```
s_evaluator *S_evaluator  = get_current_evaluator();
```

and is empty in the non-threaded case since the variable is global. In general, it should never hurt to insert this macro at the top of each C routine to be used in S.

There are two ways to think about developing code to be used in the threaded environment. The first is how does it communicate with other threads. As we have mentioned, this is rarely a problem for simple use. However, if one or more routines are executed within several threads and communicate using some variables, this should be programmed using the same synchronization paradigm and syntax as as the base source code to ensure portability. Secondly, one must consider what would happen if two S threads invoked the routine concurrently.

We turn to the first of these issues. We have attempted to support user and kernel level threads with two different APIs — Pthreads and UNIX/Solaris threads. (Additionally, we will attempt to make the environment function in a distributed manner with only minor changes to the S functions that control the tasks.) In order to do this, we have created macros that can be used in C routines which insulate the developer from the differences between the APIs. Using these, code can be compiled and linked for different concurrent implementations with transparent and simple changes to the flags in the makefile which control the compilation. Routines should therefore be developed using these macros and this high-level interface.

We offer here only a brief description of this interface. This is subject to change as we modify the existing entries and add new facilities. The header file **ThreadAliases.h** includes the relevant set of macros based on the presence of different compiler flags. Currently, the API of interest resides in the file **PthreadsAliases.h** in the directory `include` relative to the top of the source tree specified as `SHOME` .

The interface can be separated into two components - the structure types and the actions or routines. We define the standard data types such as a `mutex` or thread as macros. For example, the following table illustrates the UNIX and Pthread types and the corresponding type used in S code.

Native Type		Macro
Pthreads	UNIX	
`pthread_t`	`thread_t`	`Thread`
`pthread_mutex_t`	`mutex_t`	`MUTEX`

Similarly, we do the same for other types defined in the different APIs.

Every instance of a `MUTEX` or `COND_VAR` objects must be initialized. This is done using the `MUTEX_INIT` and `COND_VAR_INIT` macros. Similarly, the `MUTEX_DESTROY` and `COND_VAR_DESTROY`

[8]Note that this should not be followed by a semi-colon `;`

perform the relevant cleanup when the objects are not needed.[9] Locking and unlocking of a `MUTEX` structure are done via the `LOCK`, `TRY_LOCK` and `UNLOCK` macros. Similarly, condition variables are controlled with the `COND_WAIT`, `COND_SIGNAL` and `COND_BROADCAST`. Other operations such as thread creation and thread attribute handling should be done using S functions. However, if they are performed directly in C routines suitable macros can be found in the header files mentioned above. This also applies for function calls for operations such as yield, etc.

One useful approach in eliminating global variables from non-threaded is to make them thread specific. This can be done using the `SET_TSD` and `GET_TSD` macros. One creates a (hash) key with which the thread specific data can be cached in the thread's table. This is of type `ThreadKey` and created using the macro `INIT_THREAD_KEY`. Any object can be stored within a thread under this key by specifying the object's address in the call to `SET_TSD`. It is retrieved using `GET_TSD`. Then the code that retrieves such an object must cast it to an appropriate type. Objects stored in this way can only be retrieved by the thread in which the call to `SET_TSD` was made and each thread can have its own object stored using the same key.

The following example illustrates how this idea works. The idea is that we use the variable `key` as the identifier for storing the data in the routine `foo()`. The object is retrieved and used in `bar()`. While this example shows that the object can be removed from the thread's table of data, this is not necessary. The key must be global and initialized appropriately. The details should be relatively obvious from the example above but are explained in the more general setting of dealing with global or static variables in the paragraphs following the example.

```
static ThreadKey key;
static DO_ONCE_LOCK key_once = THREAD_ONCE_INIT;

void foo() {
 S_EVALUATOR  /* Needed for Calloc in threaded case */
 Object *obj =  Calloc(1, Object);

      /* See below for details */
    THREAD_ONCE(&key_once, init_key);
  /* Fill in the fields in obj */
   ........

  SET_TSD(key, obj);
}

void bar() {
```

[9]These macros do not free the object, just its internal structures.

```
  Object *obj = NULL;
    /* Make sure the key has been initialized */
  THREAD_ONCE(&key_once, init_key);

    /* Retrieve the object */
   obj = GET_TSD(key);
   if(!obj) {
     PROBLEM "bar called out of order!"
     ERROR;
   }

   /* Use obj */
    ........

   /* Tidy up after using the object */
   Free(obj);
   SET_TSD(key, NULL);
 }
   /* Initialize the key */
 void init_key() {
  key = INIT_THREAD_KEY(&key);
 }
```

The second issue of creating reentrant and thread-safe routines only arises if global or static variables are used in the routine. A common instance of this involves initializing static variables in the routine. For example, in a routine used with the `.Call()` interface, one frequently will compute the indices of the different slots in an object. This might be done in the following way in the non-threaded environment.

```
s_object *
foo(s_object *in) {
 S_EVALUATOR
 static long name_slot, id_slot, data_slot = -1;

 if(data_slot < 0) {
    /* Initialize the slot indices */
   get_class_slots("class-name", FALSE, S_evaluator,
                              "name", &name_slot,
```

```
                                  "id", &id_slot,
                                  "data", &data_slot,
                                  NULL);
    }
   ..
  }
```

This implies that the call to `get_class_slots()` will be called only once — the first time the routine is called. However, in a threaded environment two threads may invoke this routine concurrently and both evaluate the expression '`data_slot < 0`' to be true. Accordingly, both will execute the body of the conditional. In this example, the effect may be simply to reinitialize the variables with the same values. However, if any allocation is performed there, either memory may be lost and/or confusion may arise in future invocations of the routine, leading to non-deterministic behavior.

The "proper" way to implement such one-time initializations is to use the Pthread standard's `pthread_once()` routine [25]. This is used with a dedicated routine and object of type `pthread_once_t` which ensures that the routine is only invoked once. The variables being initialized must be accessible to both the initialization routine and the target routine. In the same spirit as the macros above, we provide a portable interface to this routine and structure. The data type is `DO_ONCE_LOCK` and the routine is `THREAD_ONCE()`. Also, the `DO_ONCE_LOCK` object must be initialized when it is declared using the `THREAD_ONCE_INIT` macro.

The following code illustrates how this is used.

```
static DO_ONCE_LOCK slot_initialization_mutex = THREAD_ONCE_INIT;
static long name_slot, id_slot, data_slot = -1;

   /* Routine to initialize the slots */
static void
slot_init(void) {
  get_class_slots("class-name", FALSE, S_evaluator,
                              "name", &name_slot,
                              "id", &id_slot,
                              "data", &data_slot,
                              NULL);
}

s_object *foo(s_object *in) {
    /* Make sure the slots are initialized, but only once */
  THREAD_ONCE(&slot_initialization_mutex, slot_init);
```

```
    /* Continue knowing the slots have been initialized */
     ..
     ..
}

s_object *bar(s_object *out) {
     /* Make sure the slots are initialized, but only once */
   THREAD_ONCE(&slot_initialization_mutex, slot_init);

    /* Continue knowing the slots have been initialized */
     ..
     ..
}
```

Note that the variables can now be used in more than one routine but the `slot_init()` routine is guaranteed to be invoked only once. Perhaps this is a slightly clumsy way of initializing variables but it does ensure the necessary synchronization.

A legitimate alternative to this approach in the case of initialization is to have the relevant routines invoked when the chapter containing the shared library is attached. This can be done using the S function `.on.attach()`. Similarly, any allocations performed (indirectly) by this routine can be released when the chapter is detached using `.on.detach()`. Since the attachment of the library must be completed before any thread can invoke a routine in the library, this solves the synchronization problem at the expense of unnecessary computation should the relevant variables never be referenced.

Just as a user's native code should be reentrant and multi-thread safe, so too should the standard S libraries which include the model fitting functions [13] and trellis plotting functions [3]. Our preliminary work has focussed on the base source code rather than these libraries. Accordingly, at this point in time, routines in these libraries should not be called concurrently from more than one thread. This does raise an interesting issue. How might we safely use such non-threaded libraries in a threaded environment? A simple solution is to ensure that only one thread calls routines within such a library. We can do this by dedicating a thread to be used as a server for functions and routines in this library. This evaluates calls passed to it from other threads using `sendTask()` with the `wait` argument passed as `T`. This ensures that the calls are serialized and that only one thread has access to any global or static variables. We are not proposing this as a long-term solution but do recommend it during development stages of libraries that use legacy code.

The same synchronization issue arises when using third-party libraries such as the NAG, LINPACK and other collections of routines. If these are not thread-safe, serialized access ensures safety. This avoids the necessity of modifying other peoples code. In the future, hopefully, more libraries will be created or re-created to support threaded environments.

Ensuring serialized access to routines within a library can be tedious and will not be completely safe unless done at the points of invocation. These consist of the `.Call()`, `.C()` and `.Fortran()` functions. We have a simple mechanism that will allow routines or an entire library to be registered as non-threaded with the S engine. Each invocation of a registered routine or routine within a registered library will first acquire a `mutex` dedicated to that routine. Accordingly, only one invocation can be running at a single time and the calls will be serialized. This avoids having to modify the S functions that call the routine to instead call a wrapper that serializes these by sending them to a dedicated thread or whatever. This feature will be implemented in the very near future to handle the models and trellis libraries before an attempt to make them thread-safe is undertaken. We wil extend the existing function `setInterface()` to enable it to register a routine as thread-safe or not.

5.7 Restructuring of S Internals

One of the benefits of using threads within the internal S engine is that it allowed us to restructure the code into more modular tasks. One aspect that warrants closer attention is the event loop that was used to drive the non-threaded version of S. The toplevel structure of the S process was divided into two stages. The first was the necessary initialization. The second consisted of an event loop which was constructed to support readers with input on an arbitrary number of connections, monitors or timers and sub tasks within an evaluation. This event loop allowed other event driven applications to be embedded inside the S process by treating each event loop as an S connection on which the events were delivered. Thus the event loop consisted of heterogeneous types of "events". The event loop was a significant extension of version 3 of S, allowing more elaborate features to be added and making the S process more of a system or environment. However, it was complicated and fragile in the sense that modifications to the implementation could seriously impact the run-time behavior of the different tasks. The heterogeneity of the events caused many of these problems. Additionally, there were necessary limits to the depth of the sub-tasks permitted.

Threads allow us to remove this event loop and to use a separate thread for each source of events — readers, monitors, GUIs and other constructs we have not yet considered. Each of these is handled in the same manner without the need for specialized code. The concept of a subtask can also be discarded and replaced with the `sendTask()` idea. We have not completely removed the presence of an event loop. Instead, we have replaced the heterogeneous event loop with a more structured version which processes homogeneous events. This is the evaluator manager. As we have described above, this is extensible and relatively easy to visualize, understand and maintain.

The work involved in transforming the existing code to allow kernel level threads will also be of use in the non-threaded environment. Since we can now create multiple S evaluators and pass these to the relevant routines, subtasks can be handled in a simpler fashion. For this to be effective, we must avoid the problem mentioned earlier of overflowing the size of the process' stack. Additionaly, computation of the thread's evaluator via the macro `S_EVALUATOR` must also be modified to allow a sub-task to push its evaluator onto the global variable defining the active evaluator. As that sub-task completes, it must restore the previous value of this variable.

Another modification to the internals that we were considering was the database interface. At one point, it seemed appropriate to crystallize their behavior as object servers by executing each database in its own thread with its own message queue and protocol. The basic operations — `assign()`, `get()` and `exists()` — would then be implemented in terms of messages. The serialization of operations would be guaranteed in the same way as the access to the global state strucutures accessed by the evaluator manager. We chose not to make these changes as it was feasible to embed the necessary functionality into the existing approach. However, in the context of distributed computing and the S-Java interface style of applications, it would appear that we have to move the databases from memory towards more context independent implementation. In short, processes on other machines must have access to the same database instances as other processes. This will help to solve the synchronization problem mentioned in section 5.4.

5.8 Signal Handling

The UNIX signal mechanism is a crude form of event notification. Signals allow an application to react to asynchronous events. Such events include the user suspending the application, a child process terminating or an internal error such as division by zero. A thorough description of signals and the different types that can be received by an application can be found in [37].

Extensions to S in version 4 allow the user to associate S expressions or callbacks with individual signal types, similar to callbacks associated with user actions in a graphical interface. These expressions allow the developer or user to handle signals in arbitrary ways rather than accepting the default behavior of ignoring the signal or terminating the session.

The signal mechanism is similar to a thread in that when the application receives a signal, a handler is automatically called inside a new stack. This executes the relevant instructions and returns. At this point, the original computations are resumed and the associated stack restored. There are, of course, issues of synchronizing variables use in the standard computations and those in the signal handler.

In an application based on a user level thread library, the kernel cannot deliver a signal to a thread, let alone a task within a thread. Accordingly, in the context of the multi-threaded S environment, asynchronous signals such as a signal sent from another application are delivered to the process and not the target thread. The semantics of how to deal with such a signal become somewhat vague and context dependent. For this and technical reasons associated with thread implementations, we create a thread during the S initialization process which is dedicated to handling signals. This does not receive synchronous signals (e.g. division by zero) generated during the evaluation by a thread. These are delivered directly to the offending thread.

The same user-specified expression mechanism can be employed (but is currently not) but again synchronization issues become complex. The evaluator manager would be used to insert the expression into the relevant table. If a signal is received during the time the user invokes the function to modify the relevant S handler and the time it is installed, how should this be handled? Perhaps correctly, the previous handler still in the table will be used. However, it is not clear that this will always be the case. At some point, it is possible that a signal will be handled erroneously. However, the likelihood of this is quite small. More work is needed to remove this possibility only if such a user-level signal handling mechanism is deemed useful. This is unlikely due to the presence of the reader mechanism which provides more flexible inter- and intra- process communication that can be directed to a specific thread.

In general, such difficulties with signals and the need for them should be minimized. The reason for this is that signals are commonly used as a communication mechanism between two processes or even between two threads. They are very simple in that there is only a small number of different signals available to be used and so the number of different messages is very limited. No data can be associated with these and so no context can be provided. We have discussed several different communication strategies in this and previous chapters including intra- and inter-thread and inter-process communication. These can be used in place of signals with much greater control and extensibility. Similarly, interrupting a thread can be implemented in a much simpler and user-friendly manner. See section 4.2.1 in the previous chapter for an example of how this is done.

5.9 Low Level Implementation Details

The threaded version of S requires some additional components other than the S source code to work appropriately. Some of the additional libraries are determined by the version one chooses to use — user level threads or kernel based threads. In addition to these, the other libraries, such as the GNU regular expression and the X11 libraries must also be compiled to be reentrant. The details of how this is done should be supplied in the documentation for each library. Additionally, a suitable version of the standard C and math libraries (libc and libm) may be required. Different versions of these have different "features" and some are simply unusable under Linux due to the different implementations of some of the standard routines.

We have used Provenzano's user level library [36] implementation of the Pthreads standard. This has worked relatively well and supports numerous platforms. There are a few omissions from the Pthreads standard (most notably, thread cancellation) and minor modifications are necessary to make sure it compiles and runs in the appropriate manner.[10] Other implementations of the basic Pthread interface are also available. These include the DCEThreads and FSU Threads. Provenzano's implementation is also being shipped with the GNU implementations of libc for Linux. Because of this integration with the standard other tools, this seems like the simplest library to use.

The LinuxThreads library was used to implement kernel level threads. This uses the Linux `clone()` to create threads and treats the subsequently created threads as processes that have common access to the same memory. This library requires a Linux kernel from the 2.0 series or more recent. Additionally, a suitable version of libc is required and libc 6 (also known as glibc 2) is recommended.

Solaris offers both POSIX and UNIX/Solaris threads at the kernel and user levels.

As we mentioned earlier, we have attempted to use macros to remove the dependencies on a particular thread API. Moving from Pthreads to UNIX threads or vice versa should be as simple as changing the compiler and linker flags. Time will illustrate if this is as flexible as we had hoped. In the future, we hope to be able to transparently support Microsoft threads [23] on the relevant platforms using the same mechanism.

[10] A trivial bug manifested in the `system()` call requires a simple fix.

Chapter 6

Comments and Conclusions

6.1 Summary of the Contributions of this Thesis.

We have introduced a collection of primitive classes and functions for the S environment which support all aspects of multi-threaded programming including thread creation and management and synchronization. The thread mechanism is created as a layer upon native threads for the operating system which can be either kernel or user level threads. Much of the work is modeled on, but not limited to, the POSIX threads standards. Due to the nature of statistical computations within this environment, significant differences in this work include

1. the high-level `threadLock` database locking mechanisms
2. the persistence of threads across S level tasks allowing for a simpler form of synchronization between such tasks

The S environment now behaves in some senses as an operating system in which threads can form "processes" and the databases act as file systems. The evaluator manager is effectively the kernel providing inter-process communication and other services.

Statistical applications are increasingly dealing with larger volumes of data. Spatial statistics and image analysis are two instances of this in which the volume of data available can be overwhelming. Large scale commercial applications are another source. At the same time, more complicated techniques are being used to analyze data for which no closed form solutions or formulae exist. These include (re)sampling methods such as the bootstrap, cross validation and also simulation techniques.

These two trends increase the need for increased computer resources and efficient solutions to these problems. It appears that parallel and distributed computing are the immediate solutions to this need for the foreseeable future. However, these areas have previously required a significant degree of specialization from their practitioners. Thus, users in the applied areas such as statistics have not been able to reap the rewards of the associated research. It is my belief that these features must be added to existing languages used by individuals in these areas rather than provided as generic libraries and specialized languages into which domain specific functionality, such as statistical methodology, is to be added. This thesis is an attempt to make this functionality available to a particular group — data analysts and statisticians — in a language with which they are familiar.

I believe that these extensions make S the only interpreted statistical language or environment which supports threads. As well as providing benefits for the S user community, this thesis is a vehicle for communicating the concepts of a threaded environment more broadly to the field of statistics and numerical computing. The ideas here can be extended to other similar languages such as Xlisp-Stat and Matlab — functional languages sharing important characteristics with S.

6.2 Current and Future Directions

The work we have described in the preceding chapters is almost entirely implemented in the current version of S, version 4. Minor modifications and extensions were introduced in the text that differ from the current implementation. After these have been integrated into the code base, we will continue to test the existing functionality on both single- and multi-processor environments. The testing involves both correctness and performance evaluations.

The next stage involves making the standard S chapters thread-safe so that these functions and data objects can be used in multiple threads concurrently. The Trellis library should be relatively straightforward to convert in this way. However, the models library is a more serious undertaking, if for no other reason than its size. As we proceed with the testing and the modifications to the libraries, we will probably add higher level functions that hide the low level primitives from the casual user.

As mentioned in the text, it is our hope that we can easily modify the manager to support distributed computing. To do this, we need to utilize the existing communication mechanisms, including the recently added support for sockets, and the object passing mechanism developed in the S-Java interface [10]. Perhaps the biggest change required is the mechanism by which the different processes will coordinate access to a given database. The message queue that we have used for both the evaluator manager and the `threadLockEvent` mechanism should allow simple extensions to be added to the manager without modifying the current structure of the source code. These extensions might contain the necessary synchronization requests regarding the databases. Ideas from the NOW and Condor projects should also help in providing this single meta operating system shared across the distributed machines.

We also hope to reorganize the source code for S so that we can create the evaluator manager as a module or S chapter. In this way, we could develop the distributed evaluator manager in the same manner and allow users to attach and detach managers to allow transparent shifts from parallel to distributed environments. A related topic includes the ability to migrate threads from one process to another. With this functionality, we could temporarily suspend threads and capture the state of an evaluator in an S object. The task could then be continued on a different machine by creating another thread on that machine using the S object to initialize the evaluator.

The initial motivation for extending the S environment to support threads arose from a project to add support for GUI development tools into S. A relatively complete design and implementation has been developed at this point. We will now return to this, integrating threads more completely. Similarly, the connection between Java and S [12] may be reworked by its authors using S levels threads. This might involve dedicating a thread for each client communicating with the server which effectively corresponds to having a one to one correspondence between threads in the S and Java processes.

The monitoring approach using threads mentioned in chapter 4 should allow us to develop a more traditional debugging facility in S, such as those using the C routine `ptrace()`. A debugger thread would control the evaluation of a task in another thread in some manner allowing the user to trace the progress of the different steps in the task. This avoids modifying the task expressions to insert calls to interactive S debugging functions and makes the debugging mechanism higher level and more generic.

Since S is an interpreted language, it contains a parser which can return its output to the user as an S object. Also, due to the "everything is an object" paradigm within S, task expressions can be easily decomposed from their object form into the essential components. This allows us to determine which objects are being assigned to different names in different databases and compute the set of input and output objects for each tasks. This information then allows us to classify sets of tasks as dependent or independent and accordingly group them. Related tasks that must be serialized are contained within their own group and two tasks from different groups can be evaluated concurrently. This allows us to automate the development of a GUI based solely on the callback expressions. This can be extended to recursively analyze the nested function calls in an expression in an effort to parallelize these. The literature relating to parallel compilers can be applied relatively easily at the user level given the existing tools in S. Of course, this would change the nature of S being a functional language.

In summary, threads facilitate the development of many new and exciting features within an environment such as S and we hope that this work will assist people involved in statistical and numerical computation in becoming familiar with the threaded paradigm.

Bibliography

[1] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A case for networks of workstations: Now. *IEEE Micro*, February 1995.

[2] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison Wesley, 1994.

[3] R. A. Becker and W. S. Cleveland. Trellis graphics user's guide. Technical report, Lucent Technologies Bell Labs, 1995.

[4] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language*. Wadsworth & Brooks/Cole, 1988.

[5] Adrian Biran and Moshe Breiner. *MATLAB for engineers*. Addison-Wesley Publishing Company, 1995.

[6] Graham Birtwistle, Ole-Johan Dahl, Bjorn Myhrtag, and Kristen Nygaard. *Simula Begin*. Auerbach Press, 1973.

[7] John Bloomer. *Power Programming with RPC*. UNIX Network Programming. O'Reilly & Associates, Inc., second edition, 1992.

[8] Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis. *Visual Object-Oriented Programming*. Manning Publications Co., 1995.

[9] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge, Mass., 1990.

[10] John Chambers and Mark Hansen. The S/Java interface: Programming with data on the web. URL http://cm.bell-labs.com/cm/ms/departments/sia/project/java/html/overview.html, May 1997.

[11] John M. Chambers. Greater or lesser statistics: A choice for future research. Technical report, AT&T Bell Laboratories, 1993.

[12] John M. Chambers. *Programming with Data: Version 4 of S.* In preparation, 1997. Beta Testers Draft.

[13] John M. Chambers and Trevor J. Hastie, editors. *Statistical Models in S.* Chapman & Hall, 1993.

[14] Patrick Chan and Rosanna Lee. *The Java Class Libraries: An Annotated Reference.* The Java Series. Addison Wesley, 1996.

[15] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood with incomplete data via the em algorithm. *Journal of the Royal Statistical Society, Series B*, 39:1–38, 1977.

[16] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap.* Monographs on Statistics and Applied Probability. Chapman & Hall, 1993.

[17] Scott Fields. Hunting for wasted computing power. Technical report, University of Wisconsin-Madison, 1993. http://www.cs.wisc.edu/condor/doc/WiscIdea.html.

[18] David Flanagan. *Java in a Nutshell.* O'Reilly & Associates, Inc., second edition, 1997.

[19] Bill O. Gallmeister. *POSIX. 4: Programming for the Real World.* O'Reilly & Associates, Inc., January 1995.

[20] Enden R. Gansner and John H. Reppy. *A Foundation for User Interface Construction*, chapter 14, pages 239–260. In Myers [28], 1992.

[21] Al. Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[22] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI : Portable Parallel programming with the Message-Passing Interface.* Scientific and engineering computation. MIT Press, Cambridge, Mass., 1994.

[23] M. Johnson Hart. *Win32 System Programming.* Advanced Windows Series. Addison-Wesley Developers Press, 1997.

[24] Jan Rune Holmevik. Compiling simula: A historical study of technological genesis. URL http://lingua.utdalls.edu/articles/simula.html, 19.

[25] Steve Kleiman, Devan Shah, and Bart Smaalders. *Programming with Threads.* Sunsoft Press, 1996.

[26] Doug Lea. *Concurrent Programming in Java.* The Java Series. Addison Wesley, 1996.

[27] John Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison Wesley, 1996.

[28] Brad A. Myers. *Languages for Developing User Interfaces.* Jones and Bartlett Publishers, 1992.

[29] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx-Farrell. *Pthreads Programming.* O'Reilly & Associates, Inc., 1996.

[30] Charles J. Northrup. *Programming with UNIX Threads.* John Wiley & Sons, Inc., 1996.

[31] Adrian Nye. *Programmer's Supplement for Release 6.* O'Reilly & Associates, Inc., 1995.

[32] Scott Oaks and Henry Wong. *Java Threads.* The JAVA Series. O'Reilly & Associates, Inc., January 1997.

[33] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley professional computing series. Addison-Wesley, 1994.

[34] John K. Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX Technical Conference 1996*, 1996.

[35] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterline. *Numerical Recipes in C.* Press Syndicate of the University of Cambridge, 1991.

[36] Christopher Provenzano. An implementation of the pthreads interface. URL http://www.mit.edu:8001/people/proven/pthreads.html, November 1996.

[37] W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 8th printing edition, 1994.

[38] Duncan Temple Lang. Tools for developing graphical user interfaces in S. Technical report, Lucent Technologies, Bell Labs, In Preparation.

[39] Luke Tierney. *LISP-STAT: An Object-Orienetd Environment for Statistical Computing and Dynamic Graphics.* Wiley Series in Probability and Mathematical Statistics. Wiley, New York, 1990.

[40] Anne E. Trefethen, Vijay S. Menon, Chi-Chao Chang, Grzegorz J. Czajkowski, and Chris Myers. Multimatlab: Matlab on multiple processors. URL http://www.cs.cornell.edu/Info/People/lnt/multimatlab.html, May 1996.

[41] Uresh Vahalia. *UNIX Internals: The New Frontiers.* Prentice-Hall Inc., 1996.

[42] Ajay Vohra. Multithreading. In Nye [31], chapter 4.

[43] Douglas A. Young. *Motif Debugging and Performance Tuning.* Prentice Hall, 1995.

The appendix has three components. The first offers a short description of several commonly used terms in the literature about threads and parallel computations. A hard copy of the on-line documentation for the functions and classes introduced in this thesis is then provided. This documentation is to be used as a reference or reminder of the signatures and basic intentions of the different functions and classes rather than an explanation of the philosophy underlying the thread chapter. Finally, a list of some interesting URLs are provided for the interested reader.

Appendix A

Glossary & Conventions

A good description of threads and the issues associated with programming multi-threaded applications is given in [25]. This describes the POSIX Threads (Pthreads) application programming interface (API).

- **Atomic:** An operation is said to be atomic if it cannot be interrupted while it is being executed by the operating system or another thread. This is important for threaded code since if an operation is modifying data and is not atomic, the data may be accessed by another thread while it is in an inconsistent state. Some low level operations are intrinsically atomic, others are made atomic using mutexes.

- **Call Stack:** When an application is run, a routine is called which in turn calls other routines, each behaving in the same manner creating a sequence or hierarchy of function calls. This is termed a stack, since each call is put on the top of the stack when it is current and subsequent calls push it down. When the call returns, it is removed from the top of the stack and the next call becomes active. Each element on the stack is a *stack frame* which contains the local variables and arguments within a function call and information about the return address, etc.

- **Context Switch:** A context switch is the act of making one thread active and another inactive. This involves scheduling a thread on a CPU and restoring the call stack and registers associated with the newly active thread. This can be done in a user level thread library or by the kernel if the threads are scheduled at that level.

- **Preemptive Threads:**hreads can be either preemptive or non-preemptive. Preemptive threads are controlled by another entity called the scheduler. Periodically, this interrupts the currently active thread(s) and determines which thread(s) should become active next.

Non-preemptive threads are not interrupted in this manner. Such a thread must relinquish its active status in order to allow another thread take its place.

- **Routine and Function:** Throughout this document, we differentiate between S and C level functions by referring to the latter as a *routine*, reserving the term *function* for S level objects.

- **Symmetric Multi Processor Machine:** The acronym SMP stands for symmetric Multi-Processor which indicates that threads within an application share the same memory. An asymmetric multi-processor indicates that each thread has its own memory. We do not discuss the AMP environment in this work except implicitly in the context of distributed computing.

- **Thread:** C level thread is stream of execution within a process. It embodies the state of a function call and its ancestors. In other words, it consists of the stack and all the registers and a map of how to proceed. Different threads share the same memory space of the process in which they are created and execute concurrently.

Appendix B

S User Level Documentation

`access`	Determine whether a thread has access to another thread

Usage: `access(source =, ...)`

Arguments: `source`: the thread for whom access is being queried. The default value is `self()` but this must be specified to identify the subsequent arguments.

`...`: a collection of `thread`, `threadGroup` and `threadTask` objects. This can contain nested lists of objects that are processed by `threadIds()` to produce a single level list of `thread` objects.

Value: a vector of logical values with an entry for each of the threads referenced in `...`. Since this can contain `threadGroup` objects and lists of `thread` or `threadTask` objects, the length of the return vector is not necessarily the length of the list `...`.

Examples:

```
 # suspend all threads we have access to
ts = threadIds(tB, groupB)
which = access(tA, ts)
suspend(ts[which])
```

See Also: `sendTask()`, `threadStatus()`, `threadAttributes()`, `thread()`, `thread`.

Key Words: thread, permission, communication, synchronization

Description: Determine whether the thread specified by source has access permission to the threads supplied in `...` This permission controls such things as the ability to query attributes of another thread such as the state, the task queue, the history list. Perhaps most importantly, the source thread must have access to the target threads in calls to `sendTask()` and `join()`.

The access permissions of a thread are determined by its `threadAttributes` object and can be set using the function `thread()` when it is created or at any time using `setThreadAttributes()`. The permissions are stored either as a: a) logical value b) `threadGroup` c) expression This object can be retrieved using `getThreadAttributes()` for more interesting analysis. However, this function determines whether the source thread currently has access to a thread.

`add.threadGroup`	Add one or more elements to a `threadGroup`.

Usage: `add.threadGroup(to, ...)`

Arguments: `to`: the existing group object.

`...`: a collection of `thread` or `threadTask` objects which will be added to the existing entries in `group`.

Value: Returns the new `threadGroup` object with the new elements contained. This must be assigned if used across expressions as the new elements are not added to the object in the database. Examples: group = threadGroup() for(i in 1:5) group = add(group, thread())

See Also: `threadGroup`, `thread` and `threadTask` classes, `thread()`, `apply()`, `threadIds()`, most of the thread functions.

Key Words: Thread

Description: Add one or more objects of class `thread` or `threadTask` to a `threadGroup`. The `threadGroup` class is used to contain `thread` objects and more specific objects identifying tasks within a thread, `threadTask` objects. This is a convenient mechanism used to dispatch methods on a list of threads.

`cancel`	Cancel the current task in one or more threads.

Usage: `cancel(..., wait = F)`

Arguments: `...`: a list of `thread`, `threadTask` and `threadGroup` objects that are processed by `threadIds`.

`wait`: if this is `T`, the calling thread waits to be notified by the manager that the cancellation has occurred. Otherwise, the tasks are canceled by the evaluator manager asynchronously and the call returns after the request is dispatched to the manager.

Value: a logical vector with an element for each of the `thread` objects referenced in `...`. The value of each element indicates whether the cancellation request was applied to the thread. If the calling thread does not have access permissions for a target thread, the request is not processed and the value will be `F`. If `wait` is `F`, a value `T` does not indicate that the task has been canceled, but just that the manager has forwarded the cancellation instruction. However, if `wait` is `T`, the result is the status of the action indicating whether the task was canceled.

Examples:

```
 # cancel all the tasks in the children
cancel(children(), wait=T)
```

See Also: `threadStatus()`, `access()`

Key Words: Thread, thread status, synchronization.

Description: Cancel the tasks in the threads specified in `...`. This can be used to synchronize the target threads so as to allow the calling thread guarantee a particular state.

This implementation of this function is relatively simple. It merely calls the function `threadStatus()` which performs the communication with the evaluator manager.

`children` — Retrieve the list of children thread created with a given thread

Usage: `children(thread = self())`

Arguments: `thread`: the thread for which the children are being requested. By default, this is the calling thread, `self()`

Value: A list of threads that were created with their parent being the thread specified in this call.

Examples:

```
  # generate some random points
points = bar()
for(i in 1:10) {
  thread(substitute(foo(i),list(i=i)))
}
 ..
  # now we have finished so cancel the tasks in
  # each of the children threads.
cancel(children())
```

See Also: `thread()`, `access()`, `parent()`, `self()`.

Key Words: Thread.

Description: If the target thread specified in `thread` is the calling thread, the computations are performed locally without a request to the evaluator manager. Otherwise, the manager retrieves the information from the target thread and returns it to the calling thread.

`currentTask`	Retrieve the task being currently evaluated by a thread.

Usage: `currentTask(thread)`

Arguments: `thread`: the `thread` object whose task we are querying.

Value: A `threadTask` object containing the task expression currently being evaluated by the `thread` in question.

Examples:

```
  # send a task to a thread, tA, if it has no current task.
  # (should also look at the output from taskQueue()
if(taskHistory(tA)== NULL)
  sendTask(Quote(nextTask()), ta)
```

See Also: `taskHistory()`, `lastValue()`, `taskQueue()`.

Key Words: Thread, task, task queue, Evaluator manager.

Description: This allows a user or function to determine what a given thread is currently doing. This can be used to describe the environment in a 'top'-like application or for dynamically determining the state of the environment and scheduling tasks accordingly.

The request is performed by the evaluator manager in the spirit of data hiding for thread values. The manager locks the task queue while it copies the current task. This prohibits the target thread from proceeding for a short period.

`elements`	Retrieve the elements of a `threadGroup` object

Usage: `elements(group)`

Arguments: `group`: the `threadGroup` object whose elements we want.

Value: a `list` of `thread` and `threadTask` objects that are stored in the `threadGroup`

Examples:

```
  # cancel all the threads in a group
 sapply(elements(group), cancel)   # equivalent to cancel(group)
```

See Also: `threadGroup()`, `add()`, `remove()`, `thread()`, `thread`, `threadTask`.

Key Words: Thread, task

Description: Retrieve the elements of a `threadGroup`. A `threadGroup` works much like a list but provides a useful structure and formalism to the concept of a collection of `thread` and `threadTask` objects. This allows methods to be dispatched based on the class `threadGroup`. These mostly behave much like `apply()` or `sapply()` functions which operate on each element of a structure. Similarly, `threadGroup` methods usually operate on the elements contained in the object. This function (`elements()()`) allows a user to perform these operations directly on the entries in the group where no appropriate method is available.

`getLock`	Acquire the ownership of a `threadLock` object.

Usage: `getLock(lock, condition =, try =)`

Arguments: `lock`: the `threadLock` object whose ownership is to be acquired.

`condition`: if specified, the lock is acquired and the call returns only when this condition is true. This can be one of two basic forms - a conditional expression, a threadLock event. The default is `NULL` indicating that it is ignored.

`try`: if this is `T` and the lock is already held by another thread, the call returns immediately without acquiring the lock. This is used to do other things rather than block waiting for to acquire the lock. The default is `F`

Value: a logical value indicating whether the lock was held. (Should contain the depth of the lock for this thread representing the number of recursive locks).

Examples:

```
l = threadLock()
getLock(l)
  assign("x",1:10, where=l)
yieldLock(l)


 # conditional locking
getLock(l, Quote(length(x) == 10))
  .. # perform some actions based on this fact that
yieldLock(l)
```

See Also: `threadLockEvent` classes, `yieldLock()`, `tryGetLock()`, `threadLock()`

Key Words: Thread, synchronization, threadLock, mutex, condition variables.

Description: This function attempts to acquire exclusive ownership of the `threadLock` supplied as the first argument. If another thread currently has ownership, the action depends on the argument `wait`. If this is `F`, this call must wait for that thread to release the lock and then compete with other waiting threads until it eventually acquires it. If `wait` is `T`, then the call returns immediately and indicates that the lock is not held by this thread.

If the calling thread already holds the lock, the call returns after incrementing the depth or level of the lock. This allows recursive locking by a given thread rather than leading to deadlock. The lock must be released that number of times or completely in one go using `yieldLock()`. This function also allows conditional locking based on objects assigned to the `threadLock` object. The `condition` argument is a test that evaluates to `T` or `F`. The test is evaluated while holding the lock. If the test returns `T`, the call returns. Otherwise, the lock is released and the call waits to be notified of relevant changes (assignments and removals) to the the `threadLock` object referenced in `lock`. On receipt of such notification, the same locking-test procedure is repeated.

The test is usually an S expression such as

```
Quote(x == 10)
```

or

```
Quote(finished == T)
```

Alternatively, however, one may be interested in monitoring assignments to different objects to avoid re-evaluating the condition for *all* assignments or removals. Similarly, it is better to avoid continuously evaluating the expression

```
exists("finished").
```

For these, we use objects of class `threadLockEvent` as the value for the argument `condition`.

Notes: If `try` is a number, it would be nice to wait that number of seconds trying to acquire the lock.

`getThreadAttributes`	Retrieve one or more threads' `threadAttributes`.

Usage: `getThreadAttributes(...)`

Arguments: `...`: a collection of thread related objects to be processed by `threadIds()`. If no values are specified, `self()()` is used.

Value: an object of class `threadAttributes` containing the values of the different fields for the specified thread.

Examples:
```
attrs = getThreadAttributes(tA)
setThreadAttributes(attrs=attrs,  tB)
```

See Also: `setThreadAttributes()`, `thread()`

Key Words: Thread, scheduling, access permission.

Description: This retrieves the thread attributes for a given thread. If the calling thread is the same as the target thread, the computations can be done locally. Otherwise, a request for the target thread's information is sent to the evaluator manager. This information is copied to the current thread and returned by the function call.

Note: We may add the same fields as those in `setThreadAttributes()` and `thread()` to allow individual fields to be queried. This would avoid the overhead of querying all the fields and copying the results between threads. The current semantics will be preserved however if called with no arguments other than the target thread.

`join` Wait for one or more tasks to be completed.

Usage: `join(...)`

Value: a list of the return values resulting from evaluating the different tasks being waited on.

Examples:
```
task = sendTask(Quote(foo), tB)
  ..  # do some other things
  # wait till the task is completed
join(task)



for(i in 1:k) {
   group = add(group, thread(substitute(foo(i), list(i=i))))
}
 # wait for all the children threads to complete their tasks
join(group)
```

Description: This function waits for all the tasks supplied in the argument list to be completed by the relevant thread. The arguments are objects of `threadTask` which identify the task and thread in which the task is being performed. The function communicates with the evaluator manager to determine which tasks have already been completed. Then it sleeps using few CPU resources waiting to be notified as each of the outstanding tasks are completed. The evaluator manager notifies the waiting thread of such events.

If a thread, A, attempts to join on a thread B to which it does not have access permission (see `threadAttributes`, `access()`) an error is signaled and the call is terminated.

See Also: `threadLock()`, `getLock()`, `yieldLock()`, `sendTask()`, `threadTask()`.

Key Words: Thread, synchronization, task, evaluator manager.

`kill` — Terminate one or more threads

Usage: `kill(...)`

Arguments: `...`: A collection of `thread`, `threadTask` and `threadGroup` objects that identify threads which are to be terminated.

Value: a logical vector of elements corresponding to the threads in `...` processed by `threadIds()`. If an element has the value `T`, then the calling thread has access to the corresponding target thread and the termination message has been issued.

Examples:

```
for(i in 1:k) {
 thread(substitute({foo(k);sendTask(Quote(kill(children())))},
        list(k=k)))
}
kill(children())
```

See Also: `threadStatus()`, `suspend()`.

Key Words: Thread, synchronization, status, evaluator manager.

`lastValue` — Return the `.Last.value` from a thread's working database.

Usage: `lastValue(thread)`

Arguments: `thread`: the target thread whose working database will be queried for the object `.Last.value`. If this is omitted, `self()` is assumed.

Value: the object named `.Last.value` from the the target thread's working database. See `valueOf()`.

Examples:

```
tB = thread(Quote(foo))
 ..
 # after we know it has been completed, other than using
```

```
 # join() which would return the last value!
print(lastValue(tB))
  # contrast with valueOf (performed in place of lastValue)
  # when other threads are sharing the interactive
  # thread's working databases.
print(valueOf(tB))


  # three ideas of returning the last value!
  # these are different depending on the
  # existence of other threads
> 1:10
> .Last.value
> 1:10
> lastValue(self())
> 1:10
> valueOf()
```

Description: This function is provided to provide an easy access to the object `.Last.value` for a thread which is not necessarily directly available to another thread as in the non-threaded case. Since databases and a thread's search path may be thread-specific, retrieving the object `.Last.value` for a remote thread requires identifying the thread's working database and then using `get()`. This function is `syntactic sugar` for these operations.

However, in most contexts, the meaning of this object is, at best, unclear. Since threads can share their working database, the value of `.Last.value` will depend on the order of completion of the threads' tasks. Thus there is a race condition which may lead to different results depending on many factors when replicated. Another synchronization mechanism must be used to avoid this race condition.

If a user really wants the value from the last task evaluated by a thread, use `valueOf()`. However, this is usually used interactively, but is more appropriate

See Also: `valueOf()`, `threadLock()`, `getLock()`, `yieldLock()` `taskHistory()`.

Key Words: Thread, task history.

`lockOwner`	Determine the threads holding one or more

Usage: `lockOwner(...)`

Arguments: `...`: a collection of `threadLock` objects.

Value: a list of `thread` objects corresponding to the owner. An element which is `NULL` indicates that the lock is not held by any thread.

Examples:

```
l = threadLock()
 ...
lockOwner(l)
```

Description: This allows the user to determine what thread owns a specific lock. If one or more threads are waiting for this lock, identifying the thread and its current task help diagnosing the problem.

This is and should be primarily used for debugging purposes to get a view of the current state of the session environment. Tasks should have a structured mechanism for acquiring and releasing `threadLock` ownership. Thus they should never depend on this function to determine which lock to acquire.

See Also: `threadLock()`, `getLock()`, `yieldLock()`, `searchPath()`

Key Words: Thread, synchronization, mutex.

`LockOwners` — List all locks and current owners known to the session

Usage: `LockOwners()`

Value: A list containing an element for each `threadLock` in existence in the session. Each element consists of the `threadLock` object and the `thread` which holds the ownership or `NULL` if it is not currently held.

Examples: `locks = LockOwners()`

See Also: `threadLock()`, `getLock()`, `yieldLock()`, `lockOwner()`.

Key Words: Thread, synchronization, debugging, evaluator manager.

Description: This function returns a list of all the locks in existence, coupled with the thread that currently holds it. This is used as a debugging mechanism for synchronization problems, especially diagnosing deadlock. More often, a user will have identified a subset of the locks which might be causing a problem and can use `lockOwner()` for that smaller group of locks. However, this function can be used when the locks need to be identified. Usually, this would indicate a misunderstanding of the basic thread synchronization being employed. However, this is useful for understanding other developer's code.

`parent`	Retrieve the parent threads of the specified threads.

Usage: `parent(...)`

Arguments: `...`: A collection of `thread`, `threadTask` and `threadGroup` objects that identify threads. If no arguments are supplied, `self()` is used.

Value: A list of `thread` objects each of which is the parent of the corresponding element in the list of threads from

```
threadIds(...)
```

Examples:

```
# send a task to my parent.
sendTask(Quote("I'm finished"), parent())
```

See Also: `children()`, `thread()`

Key Words: Thread, synchronization, thread hierarchy.

Description: A thread is created as the child of a given thread which is its parent. This function returns that parent thread for each of the threads specified in the `...`.

`q`	Ask the Evaluator manager to terminate the session, gracefully.

Usage: `q(n = 0)`

Arguments: `n`: optional integer value, to be used as the exit status of the S process. (In shell scripts, a non-zero exit status is conventionally an indication of an error.)

Side Effects: Causes termination of the S session and returns to the operating system. If a graphics device is active, a device-dependent wrap-up routine will be executed. The function or expression `.Last` will be called or evaluated before quitting. All the threads are notified of the pending session termination allowing them to gracefully complete their tasks quickly. This is similar to the shutdown procedure of a modern operating system. The different tasks must be given adequate time to finish so that the environment be left in a suitable state for restarting the next session.

See Also: `exit()`

Key Words: Thread, interface

`removeTask`	Remove a task from a thread's task queue

Usage: `removeTask(expression, thread, wait =)`

Arguments: `expression`: the task to be removed from the target thread's task queue. This can be either an S expression (suitably quoted) or a `threadTask` object. The latter is more efficient if obtained from the relevant call to `thread()` or `sendTask()`.

`thread`: the target thread whose task queue is to be modified.

`wait`: a logical value indicating whether the request should be performed synchronously or asynchronously.

Value: logical value indicating if the task was removed or not.

Examples:

```
task = sendTask(Quote(foo()), tB)
if(!removeTask(task, as(task,"thread"), F)) {
  stop("Can't remove task")
}
```

See Also: `sendTask()`, `thread()`, `taskQueue()`, `currentTask()`.

Description: The request is sent to the evaluator manager. The task provided by the argument is searched for in the target thread's task queue and that thread cannot process the task queue for the duration of the evaluator manager's operation. If the task is the current task, it is canceled. If it is an entry in the task queue, it is removed. Matching S expressions is relatively expensive in terms of the number of computations needed. It is suitable for most purposes. However, the `sendTask` class is used to make the comparison of tasks more efficient. Each task within a thread is assigned a unique number which is returned to the user from a call to the `sendTask()` or `thread()` in which the task is supplied. This number identifies the task expression and so can be used to match a task and avoid comparison of complex S objects.

Key Words: Thread, task, evaluator manager.

`searchPath`	Retrieve the search path of a given thread

Usage: `searchPath(thread)`

Arguments: `thread`: the `thread` whose search path is being requested.

Value: a vector of objects equivalent to a call to `search()` but for the specified thread rather than the calling thread.

Examples:

```
l = searchPath(tB)[1]
getLock(l)
```

```
assign("x", x, where=l)
yieldLock(l)
```

See Also: `search()`, `attach()`, `sendTask()`, `thread()`

Key Words: Thread, attributes, chapter, synchronization

Description: The search path is thread-specific. The call

```
search()
```

returns the search path for the calling thread. The function `searchPath()` retrieves the search path for another thread. The call requests the information from the evaluator manager which copies it to the calling frames storage area. Recall that the search path is inherited from a thread's parent at creation time. The function `sendTask()` can be used to modify the search path of another thread.

Note: Ideally this should give us back a list of databases rather than character strings. However, we must discuss this further in relation to the new database mechanism and the function `search()`.

`self`	Return the `thread` object identifying the calling thread.

Usage: `self()`

Value: `thread` object identifying this, the calling thread

Examples:

```
tB = thread()
  ..
  ..
attrs = getthreadAttributes(self())
sendTask(substitute(setThreadAttributes(attrs),
                       list(attrs = attrs)), tB)


 # send a task to itself to do things periodically.
sendTask(Quote({sleep(10);foo()}), self())
```

See Also: `threads()`, `thread()`

Key Words: Thread

Description: This is used to create a `thread` object that refers to this thread. This is used in many functions which use the calling thread as the default target thread. These do so in a uniform manner by invoking `self()`.

`sendTask` — Queue a task to be evaluated by a thread

Usage: `sendTask(expression, thread, wait =)`

Arguments: `expression`: the task to be evaluated by the target thread.

`thread`: the thread to which this task should be sent.

`wait`: indicator of whether the calling thread should wait for notification that the task has been completed by the remote thread before returning or not.

Value: The return value depends on the nature of the call. If `wait` is `T`, the task is evaluated by the other thread synchronously. In other words, the task is appended to the target thread's task queue and processed when it becomes the top-most entry. After the task is evaluated, the resulting object is returned to the calling thread which issued the `sendTask()`.

If `wait` is `F`, the return value is an object of class `threadTask` which identifies the thread, expression and the number in the remote thread's task queue which uniquely identifies the task. This can be used in calls to `join()`, etc. which require a task or implicitly converted to a `thread` by the relevant functions.

Examples:

```
  # create k idle threads
for(i in 1:k)
  th[[i]] = thread()



while(!converged) {
  for(i in 1:k) {
    tasks[[i]] = sendTask(substitute(foo(i), list(i=i)),th[[i]])
  }
  join(tasks)
}
```

See Also: `thread()`, `removeTask()`, `taskQueue()`, `currentTask()`.

Key Words: Thread, task, evaluator manager, synchronization

Description: Each thread sequentially evaluates the expressions in its task queue. These are sent to it by threads in the system (potentially including itself). All tasks are sent to the thread using this function `sendTask()`. This is a request to the evaluator manager which then inserts the task onto the target thread's task queue in a thread-safe manner.

`sendtask()` can be used as simple version of synchronization in certain circumstances due to the persistence of threads and their state across tasks. See the example above. A thread, B, might send a task to be evaluated synchronously by another thread, A, if the environment in thread A (such as the state of frame 0, the search path, etc.) was more suitable for the evaluation. This might happen if we have a dedicated thread such as a Help daemon thread which cached information locally rather than recomputed it each time, etc. If the calling thread, B, must receive notification that the task has been completed, the `wait` argument provides a simple mechanism for this.

setThreadAttributes — Set

Usage:

```
setThreadAttributes(priority =, access =, maxHistoryLength =,
                    input =, output =, error =, attrs =)
```

Arguments:

`priority`: see `threadAttributes()`

`access`: see `threadAttributes()`

`maxHistoryLength`: see `threadAttributes()`

`input`: see `threadAttributes()`

`output`: see `threadAttributes()`

`error`: see `threadAttributes()`

`attrs`: see `threadAttributes()`

Value: The previous `threadAttributes` object.

Description: Sets the specified `threadAttributes` fields for one or more threads. The fields in the argument `attrs` are overridden by the individual objects specified for the corresponding arguments in this function call. Only those values specified are set allowing the remaining fields to be left unchanged.

Examples:

```
setThreadAttributes(priority=MaxThreadPriority,
                    access=self())
```

See Also: `getThreadAttributes()`, `thread()`, `threadAttributes()`.

Key Words: Thread

Description:

sleep Temporarily suspend the execution of the thread for a specified

Usage: `sleep(t =, m =)`

Arguments:

`t`: duration in seconds for the sleep.

`m`: number of additional micro-seconds to wait. Default is `0`

Side Effects: The calling thread sleeps for the specified period. Other threads can become active in this period.

Examples:

```
# the basic monitor action which repeats
# a task periodically
while(T) {
  sleep(10, 2)
  foo()
}
```

See Also: `yield()`, `suspend()`.

Key Words: Thread, synchronization, monitor.

Description: sleep for `t` seconds and `m` micro-seconds. Returns the number of seconds slept (although actual elapsed time may be more than this). The return value is no longer portable! Linux, and few other systems, will return a value indicating the amount of time actually slept or remaining when the call to select returned.

This threaded version is slightly different from the original version in S3 or S4, because it offers greater precision due to the added argument and also it uses a blocking call to the C level routine `select()`.

suspend Suspend the evaluation of one or more threads

Usage: `suspend(...)`

Arguments: `...`: a collection of `thread`, `threadTask` and `threadGroup` objects.

Value: a vector of logical values indicating whether the calling thread had access to the relevant thread. The position corresponds to the output of

```
threadIds(list(...))
```

Examples: `suspend(children())`

See Also: `threadStatus()`

Key Words: Thread, synchronization, status

Description: Suspend a thread so that is cannot proceed evaluating its current task or processing its task queue. The thread sleeps consuming little or no CPU resources. This call should be subsequently followed by a call to `threadStatus()` to continue the evaluation.

`taskHistory`	Return a list of the previous tasks and values.

Usage: `taskHistory(thread)`

Arguments: `thread`: the target thread whose task history is of interest.

Value: a list of elements, each containing a `threadTask` object and an S object which is the result of evaluating that task. The length is the minimum of the number of tasks evaluated and the value of the thread attribute (see `setThreadAttributes`) `maxHistoryLength`.

Examples:

```
els = taskHistory(tB)
if(els[1]$value == T && els[2]$value == T) {
  # do something since both tasks were successful
}
```

Description: The history of tasks evaluated within a thread is maintained by the thread so that users can determine what the thread has done and the different outcomes of those tasks. Also, this history information is used by the function `join()`. This is allows another thread to wait until a particular task has been completed by another thread. It does this by examining the history queue of that target thread to determine if the task has already been completed. The length of the task history list is governed by the thread attribute `maxHistoryLength`. This can be modified to tradeoff the length of the history list for memory usage. The larger the value, the more elements in the history list and the greater the amount of memory used to store these elements. The default value will be generally adequate for most users. Only if an unusual number of large tasks are to be evaluated by this thread, should it be modified. When modifying this attribute, the user must ensure that subsequent calls to `join()` refer to tasks that are within this task identifier range for the history list. Otherwise, if the task has been discarded, the `join()` will never return rather than returning immediately.

See Also: `taskQueue()`, `currentTask()`, `threadTask`.

Key Words: Thread, task, evaluator manager, synchronization.

taskQueue	Return a copy of the contents of the task queue

Usage: `taskQueue(thread)`

Arguments: `thread`: the `thread` whose task queue is to be returned. If this is not supplied, `self()()` is used.

Value: A list of `threadTask` objects which are pending in the .

Examples:

```
  # migrate the queued tasks to other threads
suspend(tB)
els = taskQueue(tB)
for(i in 1:length(els)) {
  thread(as(els[[i]],"call"))
}
threadStatus(tB, "CONTINUE")
```

Description: For a remote target thread, a request is sent to the evaluator manager which returns the elements of the task queue. If the target thread is also the calling thread, the computations are done locally. It is not necessarily a good strucuture for a function or task to depend on examining its own task queue for its functionality.

The result of this function can be used to examine the role of the thread and to identify tasks that can be removed or migrated to other threads. This a useful function if a higher level (S) task scheduler is to be used.

Note: Does this also return the current task? Do we need an extra argument for this to avoid race conditions?

See Also: `sendTask()`, `removeTask()`, `threadTask`

Key Words: Thread, task, evaluator manager

thread	Create a new thread.

Usage:

```
thread(expression =, name =, parent =, start =,
       priority =, access =, maxHistoryLength =,
       input =, output =, error =, attributes =,
       toplevel =, data =, action =)
```

Arguments: `expression`: optionally specifies a task for the thread. This is usually created using `substitute()` or `Quote()`. The expression is evaluated in the new thread's environment.

`name`: the name to be used for the thread in the environment. This does not have to be unique and is used only for the purposes of annotating output.

`parent`: the `thread` object which is to be the parent. This is the calling thread, by default. Its importance comes from the fact that the new thread inherits its attributes from its parent.

`start`: logical value indicating whether the new thread should start processing its task queue immediately or be initially created in suspend mode. In the latter case (`F`), a subsequent call to `threadStatus()` must be issued to allow the thread to proceed. The default is `T`.

`toplevel`: a database to be used as the working database for the new thread. This is attached as the first element of the search path of the new thread.

`data`: a named list of objects which are stored in the thread's frame 0. The same effect can be done in the initial task.

`action`: an S function which takes a single expression and '`processes`' it. The default is equivalent to `eval()`. Each task in the queue for this thread is processed by invoking this function with the task expression as the argument.

The following arguments allow the individual fields in the `threadAttributes` to be overridden.

`priority`: see `threadAttributes`.

`access`: see `threadAttributes`.

`maxHistoryLength`: see `threadAttributes`.

`input`: see `threadAttributes`.

`output`: see `threadAttributes`.

`error`: see `threadAttributes`.

`attributes`: default is the `threadAttributes` from the parent thread.

```
threadAttributes(parent())
```

Value: If a non-`NULL` object is supplied for the argument `expression`, an object of class `threadTask` is returned and that task is inserted into the newly created thread's task queue. Otherwise, an object of class `thread` is returned which identifies the newly created S and C level threads.

Examples:

```
tA = thread()
tB = thread(Quote(foo(a)))
tC = thread(substitute(foo(a),list(a=b)))
```

```
tD = thread(parent=tA)
tE = thread(data=named(a=1,b=1:10,foo = function(x) sum(x)))
```

See Also: `threadAttributes()`, `sendTask()`, `removeTask()`, `threadTask`.

Key Words: Thread, task, evaluator manager.

Description: An S user level thread is created and is schedulable by the low level thread scheduler. If a task expression is supplied, it is appended to the new thread's task queue and becomes the initial task for the thread.

The evaluator manager is in charge of creating the thread. The new thread could be a recycled thread as cached by the evaluator manager. The advantage of this is that the overhead of creating a new C level thread and associated evaluator is reduced.

`threadAttributes`	Create a `threadAttributes` object.

Usage:

```
threadAttributes(priority =, access =, maxHistoryLength =,
                 input =, output =, error =)
```

Arguments: `priority`: an integer value between 1 and `maxThreadPriority` indicating the relative importance of the thread. This is specific to the low level thread scheduler and also dependent on the other threads in existence within the environment.

`access`: argument, The default is `T`, indicating that all threads have access to this thread.

`maxHistoryLength`: this is the number of the previous tasks and results stored by the thread. This list is used by `history()` and also the function `join()`. The larger this value, the more memory used to store the tasks and objects. However, if the length of this list is too small, the function `join()` may not work correctly. The default is 50 which should be adequate for most situations.

`input`: the connection on which input is read in functions such as `scan()`, etc.

`output`: the connection on which standard output is written by functions such as `cat()` and `print()`.

`error`: the connection on which errors and warnings are written.

Value: an object of class `threadAttributes` containing the values specified in the arguments and an identifier of which arguments were specified and which should use the default value from the context.

Examples:

```
a = threadAttributes(priority=maxThreadPriority/2, access=parent())
for(i in 1:k)
   thread(attrs = a)
```

See Also: `thread()`, `setThreadAttributes()`, `getThreadAttributes()`.

Key Words: Thread, evaluator manager, scheduler.

`threadGroup` — Create a `threadGroup` object.

Usage: `threadGroup(...)`

Arguments:

`...`: list of `thread`, `threadTask` and `threadGroup` objects to be added to the group.

Value: a `threadGroup` object containing the elements specified in the `...` argument list.

Examples:

```
group = threadGroup()
for(i in 1:k) {
 group = add(group, thread(substitute(sum(1:(i^i)), list(i=i))))
}
```

Description: A `threadGroup` is used to store a collection of thread and task related objects so that operations can be easily applied to the elements of the collection. The class is little more than a structured list which identifies its purpose to different functions allowing methods to be dispatched accordingly. In some senses, the methods act like `apply()` - operating element-wise.

Elements can be added to a `threadGroup` object using `add()` and `remove()`.

See Also: `add()`, `remove()`, `thread()`, `threadTask`, `sendTask()`.

Key Words: Thread, task

`threadIds` — Unlist a collection of thread related objects into a single list

Usage: `threadIds(x)`

Arguments:

`x`: a list of objects. Typically this is passed to the function as

```
list(...)
```

from another function.

Value: a list of elements which are of class `thread` or `threadTask`.

Examples:

```
group = threadGroup()
for(i in 1:10) {
  group = add(group,thread())
}



> length(threadIds(group, tA))
[1] 11
```

Description: This function recursively descends all of the elements in its argument, extracting individual objects of the different class types of interest - `thread` and `threadTask`. Each such object is appended to the return object which is therefore a list containing elements of these classes. This is a single level list in the sense that there are no other list containing other objects of these classes.

This function forms the basis of argument handling for most of the vectorized methods and functions in this chapter.

See Also: `thread`, `threadTask`, `threadGroup`, `add()`, `remove()`.

Key Words: Thread.

`threadLock`	Create a new `threadLock` database.

Usage: `threadLock(name =, ..., attach =)`

Arguments: `name`: name to be used for the database.

`...`: named objects to be stored in the `threadLock` after it is created. This avoids race conditions in initializing the database.

`attach`: position in the search path at which to attach the newly created database.

Value: a new database object is returned (invisibly) similar to that returned by the call

```
attach(purpose="data")
```

Examples:

```
l = threadLock()
thread(substitute({getLock(l);
                    assign("x", 1:20,where=l);
                   yieldLock(l)} ,list(l=l)))
getLock(l)
  assign("x", 1:10, where=l)
yieldLock(l)
```

See Also: `threadLock`, `attach()`, `search()`

Key Words: Thread, synchronization, mutex, condition variable, chapter, database.

`threadLockAssignEvent` — Create a `threadLockAssignEvent` object.

Usage: `threadLockAssignEvent(lock, ...)`

Arguments: `lock`: argument, no default.

`...`: collection of object names, as character strings identifying the objects to be monitored for assignments. If no objects are specified, all assignments are monitored.

Value: an object of class `threadLockAssignEvent`.

Examples:

```
ev = threadLockAssignEvent(l, "x", "y")
  # acquire the lock if x or y is assigned a value in l.
getLock(l, ev)
```

See Also: `threadLockEvent`, `threadLockExpressionEvent`, `threadLockRemoveEvent`, `getLock()`.

Key Words: Thread, synchronization.

Note: Do we need to specify the lock? Should we specify the lock- might be better to have general events?

`threadLockRemoveEvent` — Create a `threadLockRemoveEvent` object.

Usage: `threadLockRemoveEvent(lock, ...)`

Arguments: `lock`: the lock for which the event is to be used.

`...`: collection of object names, as character strings identifying the objects to be monitored for assignments. If no objects are specified, all removals are monitored.

Value: an object of class `threadLockRemoveEvent`.

Examples:

```
ev = threadLockRemoveEvent(l, "x", "y")
  # acquire the lock if either x or y is removed
getLock(l, ev)
```

See Also: `threadLockEvent`, `threadLockExpressionEvent`, `threadLockAssignEvent`, `getLock()`.

Key Words: Thread, synchronization.

`threads` Return a list of all the threads currently in existence in the environment.

Usage: `threads()`

Value: a list of `thread` objects containing all the threads managed by the evaluator manager.

Examples: `threads()`

See Also: `taskQueue()`.

Key Words: Thread, Evaluator Manager.

Description: This can be done as a tree by getting the top thread (1, as we know) and then its children and recursively obtaining the children of these. Not quick and a lot of messages and potential for race conditions. But nice simple idea. Otherwise we can ask the manager and it can dump the contents of its table.

`threadStatus` Query or set the status of one or more threads

Usage: `threadStatus(..., value =, wait =)`

Arguments: `...`: a collection of thread objects to which the operation will be applied.

`value`: if this is specified as non-`NULL`, the status of each of the threads identified in the `...` argument is set to this value. This should be a value from the vector `ThreadStatusValues`. The default is `NULL` or "missing".

`wait`: if this is `T`, the call waits for acknowledgment from the evaluator manager that each of the threads status has been changed. Otherwise, the request is sent to the manager and the call returns immediately, leaving

Value: If the operation is to query the status (no `value` is specfied), the returned object is a named vector of values indicating the status of the threads in `...`, as processed by `threadIds()`. By default, setting the status of the target threads by specifying a value for the argument `value` is performed synchronously by the evaluator manager. In other words, the function waits for a response from the manager. In this case, the return value is a vector of the previous thread status values. Each element in this vector corresponds to the thread identified in the output of

```
threadIds(list(...))
```

If the request is done asynchronously to set the values, the returned object is a vector of logical values with elements corresponding to the elements in the same call to `threadIds()` as above. Each element is `F` if the calling thread does not have permission to access the corresponding target thread. Otherwise, the result is `T`.

Examples:

```
group = threadGroup()
  ..
  ..
 # query the status of a set of threads
threadStatus(group, tA)


 # set the status of these threads to suspend
 # so they will wait to be continued before
 # proceeding with the current task evaluation
threadStatus(group, tA, value="SUSPEND")
```

See Also: `thread()`, `threads()`

Description: This method either sets or queries the status of the specified threads. The value is set by specifying a value for the argument `value`. This should be either a character string or integer from the vector `ThreadStatusValues`. This is a named list of integers and either the character string corresponding to a value or the value itself should be supplied. Note that this function is vectorized in the `...` argument for the threads, but not the `value` argument. Only a single value is used for this object.

If no value is supplied for `value`, the operation queries and returns the status of the specified threads.

Key Words: Thread, synchronization, evaluator manager.

`tryGetLock`	Attempt to acquire the `threadLock` without waiting.

Usage: `tryGetLock(lock, condition =)`

Arguments: `lock`: the `threadLock` in question.

Value: a logical object indicating whether the ownership of the lock is held by this thread or not.

Examples:

```
l = threadLock()
while(!done) {
    if(tryGetLock(l)) {
       .. # do something with the lock
      yieldLock(l)
    } else {
      # do something else.
    }
}
```

See Also: `getLock()`, `yieldLock()`.

Key Words: Thread, synchronization, mutex.

Description: This is an interface to the function `getLock()` using the `try` argument of that function.

`valueOf`	the value of the last completed task by a given thread

Usage: `valueOf(thread)`

Arguments: `thread`: the `thread` whose last task return value is being queried.

Value: an object returned from evaluating the last task completed by the specified `thread`.

Examples:

```
tA = thread(Quote(x = foo()))
..  # do something else while tA is evaluating
    # x = foo()



 # now retrieve the value of the task - x
x = valueOf(tA)
```

See Also: `.Last.value()`, `lastValue()`

Description: This is the threaded counterpart of the object `.Last.value`. Since that object is written to the working database which might be shared by other threads, a race condition exists for the object stored in `.Last.value`. This function `valueOf()` refers to the thread-specific value. It returns the object from the last completed task evaluation. The user should know what the task was if retrieving the `valueOf()` from the thread. Since other tasks can be sent to the thread, it is not necessarily clear which task's object one is getting. Thus, one must use some other form of task synchronization. The evaluator manager avoids the low level race conditions by ensuring it has exclusive access to the relevant low level object.

Key Words: Thread, task

`yield` — Voluntarily give up access to the CPU for a thread

Usage: `yield()`

Value: NULL. This is used for its side effect in controlling the lower level native threads.

Examples:

```
for(i in 1:10) {
  print(i)
  yield()
}
```

See Also: `threadStatus()`, `getLock()`

Key Words: Thread, synchronization

Description: This function causes the scheduler to potentially give the CPU currently used by this thread to another thread. This is used to distribute the resources more equitably between threads. However, one cannot guarantee that the scheduler will make a specific other thread active. If such synchronization is required, one must use other mechanisms such as `barrier`s or `semaphore`s.

`yieldLock` — Release the ownership of a `threadLock` object.

Usage: `yieldLock(l, all =)`

Arguments:

`l`: the `threadLock` object whose ownership is to be relinquished or the database index in the search path.

`all`: argument, default = `F`. If this is `T`, then the complete ownership of the lock is relinquished, regardless of the

Value: logical value indicating whether the operation was successful. `F` indicates an unsuccessful operation probably due to the fact the the lock was not held by the calling thread. (See note below).

Examples:

```
l = threadLock(l)
attach(l,1)
thread(Quote({ x = foo();
               getLock(1);
               assign("x", x, where = 1);
```

```
                yieldLock(l)
              }))
  ..
 # wait for other thread to assign x
getLock(l, "x")
   ..  # do something with x.
yieldLock(l)
```

See Also: `getLock()`, `tryGetLock()`, `threadLock()`

Key Words: Thread, synchronization, mutex, critical section.

Description: The `threadLock` object implements a recursive locking procedure. This allows the same thread to acquire the lock multiple times. However, to release the lock so that another thread can acquire it, the thread must release the lock an equal number of times. Thus, `getLock()` and `yieldLock()` must be called in nested pairs. If however, one wishes to release the lock completely in one step (for example, when an error is detected), one can use the value `T` for the argument `all`.

Note: This should return the current level of the lock if it is non-zero indicating how many times the lock must be released to fully release ownership.

Appendix C

Related URLs

- `http://www.cs.cornell.edu/Info/People/lnt/multimatlab.html`
  Matlab on multiple processors.

- `http://www-ocean.tamu.edu/~baum/linuxlist-m.html`
  Modula-3 multi-threaded, etc. MzScheme

- `http://linas.org/linux/threads-faq.html`
  Linux Threads Home Page

- `http://pauillac.inria.fr/~xleroy/linuxthreads/`
  LinuxThreads

- `http://www.nag.co.uk/`
  NAG libraries.

- `http://www.icsi.berkeley.edu/~sather/`
  Sather

- `http://www.icsi.berkeley.edu/~sather/psather.html`
  pSather - parallel Sather

- `http://www.epm.ornl.gov/pvm/pvm_home.html`
  PVM. `http://www.netlib.org/pvm3`

- `http://now.cs.berkeley.edu/`
  Network of Workstations Project.

- `http://www.cs.wisc.edu/condor/`
  Condor Distributed Computing Project

Index of S Classes

Index of S Functions