Sun Technical Bulletin

January 1993

Technical Information Services

Sun Microsystems Computer Corporation
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: 812-9301-01
Issue 1993 - 01

The *Sun Technical Bulletin* is distributed to customers with software/hardware or software-only support contracts. Send comments or corrections by electronic mail to *stb-editor@Sun.COM*, or by mail to:

Sun Technical Bulletin
Sun Microsystems Computer Corporation
2550 Garcia Avenue , M/S MPK02-04
Mountain View, CA 94043-1100 U.S.A.

Contents

1. STB Editor's Introduction **1-1**

STB Overview 1-1

2. SunOS **2-1**

Realtime Scheduling in SunOS 5.0 2-1

Introduction 2-1

Details of Implementation 2-3

Lessons Learned 2-21

Performance Measurement 2-22

Future Work 2-24

Summary 2-26

Acknowledgments 2-26

Biographies 2-26

3. Languages **3-1**

SPARCworks and the Software Development Cycle 3-1

Conceptualizing 3-2

Coding 3-3

Building Executables and Libraries 3-3

Debugging 3-4
Merging Source Code 3-4
Testing and Performance Tuning 3-4
Maintaining Software 3-5
System Requirements 3-5
SPARCworks Manager 3-6
Extending the SPARCworks Manager Palette 3-7
Drag-and-Drop Startup 3-10
Session Control 3-11
SourceBrowser 3-12
SourceBrowser as Static Analyzer 3-12
Graph and Browser Windows 3-13
SourceBrowser Base Window 3-14
SourceBrowser Capabilities 3-15
SourceBrowser Menus and Commands 3-16
Using SourceBrowser 3-18
SourceBrowser Data Flow Overview 3-28
Structure of SourceBrowser Database 3-29
Handling Large Projects 3-30
SourceBrowser Resource Requirements 3-30
CallGrapher 3-31
ClassGrapher 3-34
ClassBrowser 3-36
How SourceBrowser, the Graphers, and ClassBrowser Interrelate 3-38

4. Software Release Versions **4-1**
Current Sun Software Products and Release Versions 4-1
Operating Systems 4-1
Window Systems & Desktop Utilities 4-1
Graphics Software 4-2
Software Development 4-2
System Management 4-3
Connectivity Products 4-3
Other Products 4-4
Updated Sun Operating System Support Matrix 4-5
5. Notes & Comments **5-1**
World Hotlines 5-1
Reporting Bugs 5-3
Submitting Software Bugs: United States and Canada . . 5-3
Submitting Software Bugs: Europe 5-4
Submitting Software Bugs: Intercon. 5-8
Your STB Subscription 5-10
Index

STB Editor's Introduction 1

STB Overview

This month's STB contains articles of general interest in the topics outlined below.

SunOS

"Realtime Scheduling in SunOS 5.0" is a reprint, with updates and corrections, of a SunSoft USENIX technical paper, describing the fundmental mechanisms in Solaris 2.*x* to provide realtime scheduling functionality. The paper discusses threads and their synchronization objects; various priority values, priority inheritance, and the priority inversion problem; and scheduling attributes, including new scheduling interfaces.

Languages

This month, we reproduce three chapters from the SunPro manual on the SPARCworks product, *Introduction to SPARCworks.*

"SPARCworks and the Software Development Cycle" introduces the SPARCworks suite of development tools (SPARCworks Manager, SourceBrowser, Debugger, Analyzer, FileMerge, and MakeTool) and describes their use in the development and maintenance of application software. "SPARCworks Manager" examines the tool that manages the entire suite of tools, and "SourceBrowser" details the SPARCworks tool that assists the developer in statically analyzing source code.

Software Release Versions

This section includes the current Sun software products and release version tables. We also include a revised updating of Sun's support matrix for Operating System release levels and platforms.

Notes & Comments

World Hotlines

For Sun customers world-wide served by your local service groups, use the customer service telephone numbers listed in this monthly item. Also, look to this section during the upcoming year for details on your local support call policies and procedures.

Reporting Bugs

The information in this monthly section continues to be expanded as Sun software service centers are added world-wide. The section includes a list of Sun service centers, addresses, e-mail hotlines, telephone hotlines, and FAX numbers.

Your STB Subscription

This notice is published monthly, giving customers useful information regarding ordering multiple STB hardcopy subscriptions, and duplicating additional STB copies from a single "master" copy. This duplication permission is limited, as detailed in the note.

We would like to encourage our readership to e-mail us any comments or suggestions regarding the content of the Sun Technical Bulletin, including requests for articles on specific subject matters. We are aiming at a broad spectrum of topics that are of interest to all sectors of our audience, both highly experienced technical people and relatively inexperienced users who are new to desktop workstations. We are not the appropriate channel for researching any and all technical problems that may come up in the field, but we will do our best to follow up on questions about articles that we have printed in the recent past.

Please let us know what you think, and we will do our best to accommodate you in future issues.

Thanks.

The STB Editor
sun!stb-editor or *stb-editor@Sun.COM*

Realtime Scheduling in SunOS 5.0

By – Sandeep Khanna, Michael Sebrée, John Zolnowsky – SunSoft, Inc.

The following article is an updated version of a SunSoft technical paper presented at USENIX, Winter 1992, San Francisco, California.

Abstract

We describe the fundamental mechanisms in SunOS 5.0 to provide realtime scheduling functionality. Our primary goal was to provide bounded behavior for dispatching or blocking threads. To achieve this goal we have modified the kernel to be *fully preemptive*, guaranteeing dispatch after both synchronous and asynchronous wakeups. We have also worked toward controlling *priority inversion* in the kernel. The result is a kernel capable of delivering realtime scheduling and bounded response to a large class of user-level applications.

Introduction

A realtime operating system may be defined as one that has the ability to provide a required level of service in a bounded response time. To achieve a bounded response time, time-critical applications require control over their scheduling behavior. Increasing numbers of new, interesting applications for the desktop possess time-critical aspects. Some examples that come immediately to mind are in the area of user interfaces, multimedia, and virtual reality. These applications are typically "mixed-mode"—that is, they are partitionable into schedulable entities, some but not all of which require realtime response. It seems desirable to provide a standards-conformant, full-featured environment like SunOS for the tasks without realtime requirements. But to support these applications, SunOS must be able to provide some realtime capability to those tasks that are time-critical. From our desires to provide realtime capability and to use SunOS as the basis of our work, we derived the set of requirements listed below.

- The scheduling of tasks in the kernel should be deterministic. By deterministic scheduling, we mean the kernel should provide priority-based scheduling for user tasks, so that the time-critical application developer has control of the scheduling behavior of the system; the kernel should provide bounded dispatch latency, so that time-critical user tasks are not subjected to unexpected and undesirable delays; the kernel should be free from unbounded priority inversions.
- No draconian demands should be placed on application behavior in order to obtain realtime response. This is important for mixed-mode applications, including non-realtime components which require the general services of a UNIX environment.
- The resultant operating system should be appropriate for multiprocessor machines.
- The resultant operating system should present a standard interface to the programmer and user. In particular, the interface that we must support is that described in the System V Interface Definition.[1]

Historical implementations of UNIX have not provided bounded dispatch latency. The principal failure of these systems was that a process executing in the kernel was not preemptible. A low-priority process would retain control of the processor until the process either blocked or attempted to return to user state.

One possible solution was the use of preemption points whereby, at various points throughout the kernel, code was inserted to check if the current process should be preempted and, if so, force the preemption.[2] These preemption points could be inserted, however, only where the process could recover from

1. *System V Interface Definition, 3rd Edition*, AT&T, 1989.

2. *UNIX System V Release 4 Internals Student Guide*, AT&T, 1990.

the effects of the preemption. For instance, an exiting process releasing its memory resources could not be preempted if its process state was not safe for scheduling. Hence, while a kernel with preemption points might be able to provide bounded dispatch latency, it might be guaranteed only while no process used certain kernel facilities.

SunOS 5.0 has a fully preemptible kernel, based on fully synchronized access by kernel code to kernel storage and resources. The elimination of the pervasive use of elevated processor interrupt levels to mask interrupts leaves a small set of non-preemption intervals. This permits immediate preemption when a higher priority task becomes runnable. Another relevant feature of SunOS 5.0 is the use of dynamically loaded kernel modules to enhance kernel extensibility.

The rest of this paper consists of the following sections:

- *Details of Implementation*, which describes the objects and methods involved in implementing realtime scheduling in the kernel
- *Lessons Learned*, where we discuss some of the problems of our implementation
- *Performance Measurement*, where we describe the results of our work and the measurement methods used
- *Future Work*, where we discuss areas for possible improvement and enhancement

Details of Implementation

Several elements compose the realtime scheduling features of SunOS 5.0. The foremost of these is that the internal architecture of SunOS is based on threads.[3] Within the kernel, threads interact through the use of shared memory and synchronization objects.

The user and programmer interfaces for realtime scheduling are those provided by System V Release 4 (SVR4). Runnable threads are queued in a system-wide dispatch queue array, and the scheduler determines when and which threads are to be dispatched for execution on the system processors. We describe below the scheduler activities in kernel preemption, multiprocessor scheduling, and the techniques we used to lower dispatch latency.

3. M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, *SunOS Multi-thread Architecture*, USENIX, Winter 1991, Dallas, TX [hereafter referred to as “Powell 1991”]. See May 1992 STB for excerpts.

Elements

Figure 2-1 illustrates the scheduling elements of SunOS 5.0. The fundamental unit of scheduling in this system is the thread. A thread is a single flow-of-control. Each thread possesses a register state and a stack. The system associates with each thread extra state information relating to its schedulability. These include the thread dispatch priority and processor affinity that determines on which processors a thread can execute. In SunOS 5.0, processor affinities are not user-accessible; the default is to execute on all processors.

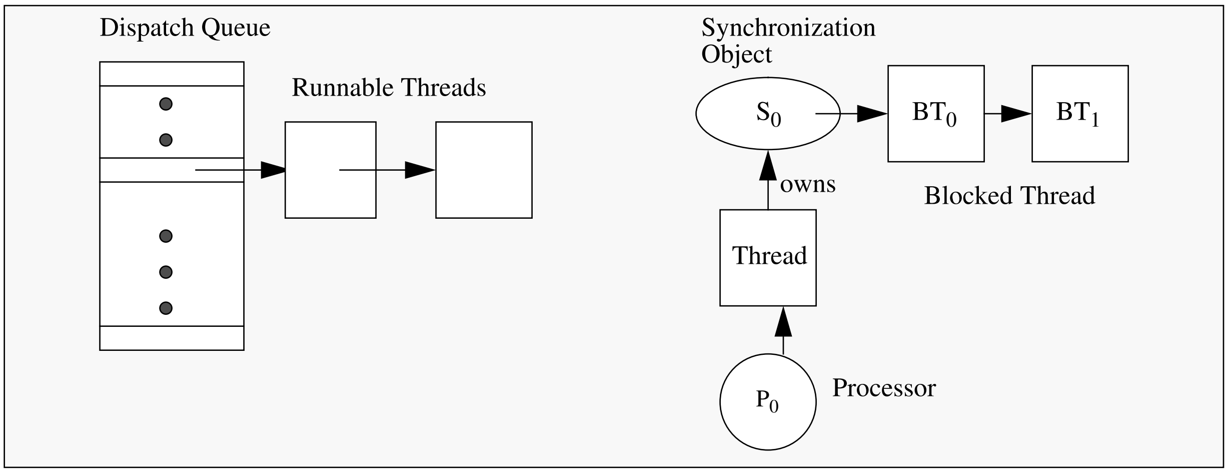


Figure 2-1 Scheduling Elements

The primary example of a thread is a lightweight process (LWP) executing within a process.[4] Such a thread has extra associated information for accessing, for example, file descriptors, user credentials, and signal context. Kernel daemon threads are responsible for pageout, swapping, and the background servicing of STREAMS. An idle thread is selected for execution whenever no other thread is runnable, and switches whenever another thread becomes runnable.

Threads interact using synchronization objects. A *mutex* (mutual exclusion lock) allows a single thread access to the information protected by the mutex. A *condition variable* allows threads to suspend execution, waiting for some change in the condition which the thread requires. A *readers/writer lock* allows a single thread to modify or multiple threads to examine some shared information. A *counting semaphore* permits synchronization from a thread that

4. Powell 1991.

does not wish to block. Operations on synchronization objects include acquisition, which may involve blocking the calling thread; and release, which may involve unblocking other threads.

Interrupt processing is performed by interrupt threads, created at interrupt time and exiting when interrupt processing is complete.[5] One particular interrupt thread, the clock thread, is dispatched upon clock interrupt, and is responsible for time-based scheduling activities. Currently, interrupt threads have an affinity for the processor which took the interrupt, and so cannot migrate to another processor. Interrupt and foreground threads use synchronization objects to interact, and hence the pervasive use of elevated interrupt level in the kernel is eliminated. *Asynchronous wakeup* refers to the occasion when an interrupt thread releases a synchronization object and unblocks another thread; we use the term *synchronous wakeup* when this is done by a foreground thread.

Priority Model

We associate with each thread a number of priority values: *dispatch priority*, *global priority*, *inherited priority*, and typically, a *user priority*. The user priority, together with other application parameters, determines the thread's global priority. The inherited priority (derived through priority inheritance as described below) and the global priority determine the dispatch priority, the actual value used in queuing and selecting a thread for execution.

The dispatcher, illustrated in Figure 2-1, uses an array of dispatch queues, indexed by dispatch priority. When a thread is made runnable, it is placed on a dispatch queue, typically at the end, corresponding to its dispatch priority. When a processor switches to a new thread, it always selects the thread at the beginning of the highest priority non-empty dispatch queue. Threads may not change dispatch priority while on a dispatch queue; the thread must first be removed and its dispatch priority adjusted, and then the thread may be placed on a different dispatch queue.

When a thread needs to wait on a synchronization object, it is placed on a *sleep queue* associated with the synchronization object. The sleep queue is maintained in dispatch priority order, so that when the synchronization object is released, the highest priority thread waiting for the object is at the head of the sleep queue.

5. J.R. Eykholt, S.R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams, *Beyond Multiprocessing: Multithreading the SunOS Kernel*, USENIX, Summer 1992, San Antonio, TX [hereafter referred to as "Eykholt 1992"]. See August 1992 STB.

Scheduling Attributes

In SunOS 5.0, threads are divided into scheduling classes. Each class chooses the attributes for the priority of threads in that class. These attributes are determined in the class-dependent functions supplied by each scheduling class. Scheduling class-dependent code must abide by certain rules expected by the class-independent code, such as that the higher the priority value, the higher the priority of the thread. However, scheduling class-dependent functions have the flexibility to decide the range of priority values for threads belonging to the class, and the class-dependent functions also determine when (if ever) a thread's priority value changes.

A new thread inherits the scheduling class of its parent. Associated with each thread is a class ID (`t_cid`) and a pointer to the class-specific data (`t_cldata`). Within the class-specific data is the class-specific priority, the time quantum associated with the thread, and the other class-related data. A thread may change its scheduling class by using the `priocntl(2)` system call. The `priocntl(2)` system call may also be used to change other parameters associated with thread processor usage.

SunOS 5.0 by default supports three scheduling classes:

- The **time-sharing** class supports a time slicing technique for threads using the processor. Time-sharing class threads are scheduled dynamically, with a few hundred milliseconds per time slice. The time-sharing scheduler switches context in round-robin fashion often enough to give every thread an equal opportunity to run.
- The **sys** class exists for the purpose of scheduling the execution of special system threads and interrupt threads. Threads in the sys class have fixed priorities established by the kernel when they are started. It is not possible for a user thread to change its class to sys class.
- The **realtime** class supports a fixed priority technique of processor access. Realtime threads are scheduled strictly on the basis of their priority and the time quantum associated with them. A realtime thread with infinite time quantum runs until it terminates, blocks or is preempted.

Figure 2-2 shows the default configuration of scheduling classes in SunOS 5.0.

Interrupt thread priorities are computed such that they are always the highest priority threads in the system. If a scheduling class is dynamically loaded, the priorities of the interrupt threads are recomputed to ensure that they remain the highest priority threads in the system.

Each scheduling class has a unique scheduling policy for dispatching threads within its class, and a set of priority levels which apply to threads in that class as illustrated in Figure 2-2. A class-specific mapping translates these priorities into a set of global priorities. The user can configure the ranges as well as the global mapping associated with each class.

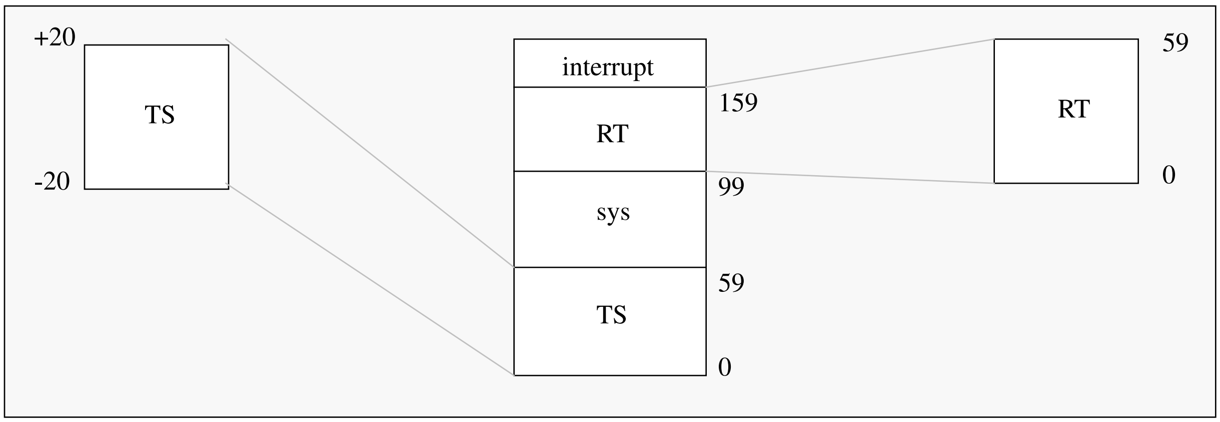


Figure 2-2 The Global Priority Model

Scheduling

SunOS 5.0 is designed to run on a shared-memory multiprocessor system. The set of threads, threads' data, and synchronization objects are shared by all processors, and the system has a single dispatch queue for all processors. We assume that each processor can send an interrupt to any other processor. Except for specially configured threads bound to a single processor, threads may be selected for dispatch on any processor.

As far as scheduling is concerned, threads can be in one of three states: *blocked*, *runnable*, or *executing*. Figure 2-3 shows examples of each of these cases. Blocked threads are those waiting on some synchronization object, such as BT_n. When the object is released, the highest priority or all waiting threads in the sleep queue are made runnable. Unblocked threads are placed at the end of the dispatch queue for their dispatch priority, such as RT_n. The thread enters the executing state when a processor selects it for execution, such as DT_n. An executing thread may become a blocked thread by waiting on a synchronization object, or may be preempted by a higher priority thread and placed back on the dispatch queue as a runnable thread. When an executing thread blocks, the system dispatches another thread.

In a uniprocessor system, realtime scheduling is defined to mean that the highest priority thread is dispatched, within a bounded time of its becoming runnable. The obvious extension to a *n*-processor environment is that the *n* highest priority processes should be dispatched. Unfortunately, this state is not achievable in a system in which some threads are restricted to or from certain processors. We have made a more local generalization, emphasizing scheduling operations in terms of a single thread or single processor at a time. We chose the following design objective: a runnable thread will be dispatched if it has higher priority than some thread currently executing on a processor for which this thread has affinity. Stated from the processor point of view: every processor is executing a thread with at least as high dispatch priority as the highest among the runnable threads having affinity for this processor. This objective can guarantee the dispatch latency for a thread only if it remains the highest priority thread on the dispatch queue until dispatched. Each arrival of a higher priority thread while a thread is awaiting dispatch restarts its dispatch latency interval.

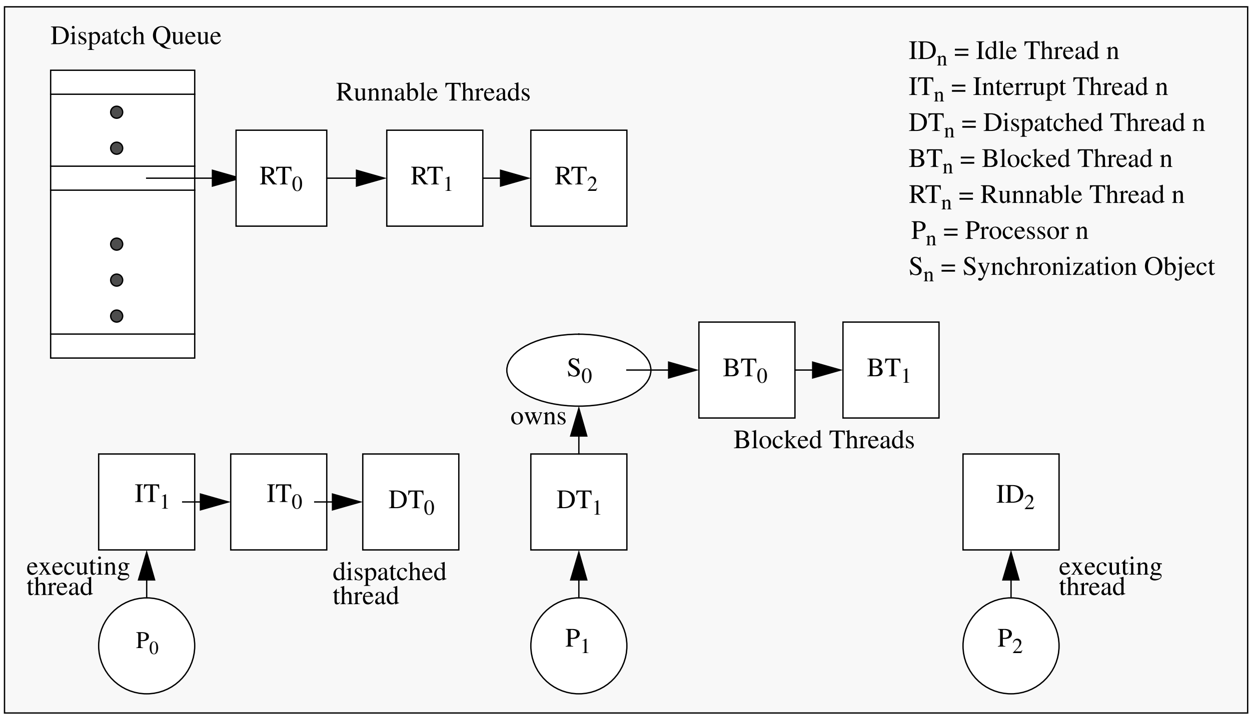


Figure 2-3 Scheduling Example

Currently, a single spin lock, `schedlock`, protects all scheduling operations. In particular, whenever the release of a synchronization object makes some thread runnable, `schedlock` is held while placing the thread on the dispatch queue. The function **`sched_lock()`** obtains this lock, and **`sched_unlock()`** releases the lock. If the lock is currently held, the processor will spin on the lock, waiting for access. To prevent interference and delays from interrupt routines, the holder of `schedlock` runs at an elevated processor interrupt level.

Associated with each processor is a set of scheduling variables: `cpu_thread`, `cpu_dispthread`, `cpu_idle`, `cpu_runrun`, `cpu_kprunrun`, and `cpu_chosen_level`.

- The `cpu_thread` value refers to the thread currently executing on the processor, and is changed whenever execution of a different thread begins.
- The `cpu_dispthread` value records the identity of the thread last selected for dispatch on that processor.
- The `cpu_idle` value refers to a special idle thread allocated for this processor, having a priority lower than any dispatch priority, and never appearing in the dispatch queue.
- The `cpu_runrun` and `cpu_kprunrun` values record requests for preemption of the current thread.
- The `cpu_chosen_level` records the priority of the thread which is slated to preempt the thread currently executing on that processor.

Table 2-1 Scheduling Interfaces

Function	Description
`setfrontdq()`	Put a thread onto the front of the dispatch queue.
`setbackdq()`	Put a thread onto the back of the dispatch queue.
`cpu_choose()`	Determine a processor to execute a thread.
`cpu_surrender()`	Have a thread give up its processor.
`disp()`	Select a thread for execution from the dispatch queue.
`swtch()`	Select the next thread to execute.
`kpreempt()`	Attempt to preempt the kernel.
`kpreempt_disable()`	Disable preemption for a critical interval.
`kpreempt_enable()`	Reenable preemption.

Threads are placed on the dispatch queue by one of two functions: **setfrontdq()** or **setbackdq()**.

- The **setfrontdq()** function is used primarily when a thread is preempted, to place it at the head of its dispatch queue so that when a thread is next chosen for dispatch from its level, it will be the first selected.
- The **setbackdq()** function places a thread at the tail of its dispatch queue. Thus a newly-awakened thread will be dispatched only after all other threads at its dispatch priority have been dispatched.

After **setfrontdq()** or **setbackdq()** has put a thread on the dispatch queue, the **cpu_choose()** function is called to find a processor on which the runnable thread might be dispatched. This function finds the processor with the lowest priority dispatched thread. If this thread has a lower priority than the newly runnable thread, that processor is marked for preemption, that processor's `cpu_chosen_level` is set to the new thread's priority, and if necessary, the interprocessor interrupt is sent. The `cpu_chosen_level` is used by later calls to **cpu_choose()** as an indication that a higher priority thread is intended for this processor.

When a thread which is currently executing on a processor has its dispatch priority lowered below that of the highest priority runnable thread, either through the effect of a **priocntl()** call or by loss of inheritance (see section below on "The Priority Inversion Problem"), it is appropriate for that thread to be preempted by the higher priority thread. The **cpu_surrender()** function accomplishes this, finding the processor on which the target thread is executing and marking it for preemption, and sending the interprocessor interrupt, if necessary. The level of preemption is determined by comparison with a system parameter, `kpreemptpri`, with lower levels causing user preemption and higher levels causing kernel preemption.

Before we can describe the dispatch operation, some details concerning interrupt thread creation and termination are required. For efficiency, the dispatch of an interrupt thread is not a complete dispatch operation. Instead, the thread executing at the time of the interrupt is pushed onto a LIFO list, and execution of the interrupt thread begins immediately, with `cpu_thread` set to refer to the interrupt thread. Since the pushed thread cannot be dispatched until the interrupt thread terminates or blocks, the interrupted thread is called "pinned." If the interrupt thread terminates without blocking, as would typically be the case, the head of the interrupt list is popped off for execution.[6]

6. Eykholt 1992.

The **swtch()** function provides the fundamental operation of scheduling. If the calling thread is an interrupt thread with a pinned thread, the pinned thread is unpinned and execution is switched to that thread. Otherwise, the function **disp()** is called to select the highest priority thread eligible for execution on this processor. If no runnable thread exists for this processor, **disp()** returns the idle thread for this processor (ID_2 in Figure 2-3). The **disp()** function automatically resets the `cpu_chosen_level` and the `cpu_kprunrun` and `cpu_runrun` preemption flags, and sets `cpu_dispthread` to reflect the newly dispatched thread.

SunOS 5.0 has two modes of preemption: user and kernel. User-level preemption refers not to a user-level request, but to a "lazy" preemption which is deferred until the thread last dispatched attempts to return to user mode; this corresponds to the historical notion of *runrun*.[7] User-level preemption forces a **swtch()** call before resuming user state execution of a process; this level of preemption is requested by setting the `cpu_runrun` flag associated with the processor. User preemption requests are recognized at the end of trap or system call processing.

Kernel preemption requests an immediate **swtch()** call, and is requested by setting the `cpu_kprunrun` flag. These requests can occur because of scheduling operations on either the same or another processor. The requests on the same processor may occur while executing the thread to be preempted, or may occur while executing an interrupt thread pinning the thread to be preempted. Requests occurring on another processor result in an interprocessor interrupt, and thus can be processed like preemption requests occurring during local interrupts.

Thus, kernel preemption requests need only be checked at two places: in the **sched_unlock()** routine, and at the end of interrupt processing. In both instances, if the `cpu_kprunrun` flag is set, the **kpreempt()** routine is called. This routine determines whether circumstances are appropriate for preemption, or should be deferred. Examples of circumstances under which preemption is deferred include when the executing thread is already slated to call **swtch()**, the calling thread is the idle thread or an interrupt thread, or preemption is disabled. The latter condition arises when some processor state, such as floating point or memory management context, is in flux. These

7. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley, 1989 [hereafter referred to as "Leffler 1989"].

conditions are bracketed by calls to **kpreempt_disable()** / **kpreempt_enable()**, with the latter function checking for a deferred preemption.

The thread time quantum is enforced by class-specific code, which marks the class-specific data to indicate an expired thread, and requests preemption. When the actual preemption occurs, the class-specific code checks the expiration mark, and if so, calls **setbackdq()** rather than **setfrontdq()** when placing the current thread on the dispatch queues, thus providing round-robin scheduling.

Priority Inversion

Priority inversion is the condition that occurs when the execution of a high priority thread is blocked by a lower priority thread. If the duration of priority inversion in a system is unbounded, it is said to be uncontrolled. Uncontrolled priority inversion can cause unbounded delays during blocking, resulting in missed deadlines even under very low levels of processor utilization. During the design phase, we identified two types of priority inversion as particular areas of concern. These areas are *hidden scheduling* and the priority inversion problem associated with the use of synchronization objects, usually called simply the *priority inversion problem*.

Hidden Scheduling

We define hidden scheduling as that work done asynchronously in the kernel on behalf of threads without regard to their priority. One example is the traditional model of STREAMS processing. In this traditional model, whenever a process is about to return from the kernel to user mode, the kernel checks to see if there are any requests pending in the STREAMS queues, and if so, these requests are processed before the thread returns to user mode. In effect, these requests are being handled at the wrong priority.

Another example is the processing of the callout-queue, a mechanism for scheduling delayed processing of specified functions. The **timeout()** function puts requests on the callout-queue; its interface is specified in Table 2-2. Regardless of whether a request for delayed processing was issued by a timesharing thread, a realtime thread, or a system thread, callout processing is done at the lowest interrupt level. It is possible that the time for a request issued by a time-sharing thread arrives while a realtime thread is running. This will cause the realtime thread to be interrupted, resulting in priority inversion.

Since neither the number nor the duration of the functions on the callout-queue are predictable, the duration of priority inversion is non-deterministic, and hence is uncontrolled. Measurements taken under SunOS 4.0.3 and SunOS 4.1 have shown that it could take longer than five milliseconds to process the callout-queue. Such unscheduled delays are unacceptable in an operating system that purports to offer realtime response.

Table 2-2 Timeout Interfaces

Interface	Description
`timeout` (*func*, *arg*, *t*)	Schedule the function ***func*** to be called at time ***t*** from now. Processed at the highest priority for system threads.
`realtime_timeout` (*func*, *arg*, *t*)	Schedule the function ***func*** to be called at time ***t*** from now. Processed at the lowest software interrupt level.

In the ideal case, all work done in the kernel would be done at the priority of the thread that requested the work. To do this for the callout mechanism would entail associating a priority with each request and creating a thread at that priority to do the requested work. The obvious drawbacks of this approach are the creation of many threads, more context switches, and the major restructuring required of the existing callout-queue framework. Another possible alternative is to process only the smaller requests at interrupt level and process the remainder at a lower priority. This approach requires complete knowledge of the entire system. With dynamically loadable modules, this is not a feasible solution.

To address the problem of hidden scheduling, we have whenever possible followed a policy of moving kernel processing of this sort into kernel threads, so that the work does not preempt or run at the expense of realtime work on the system. The processing of the STREAMS queues has been dealt with in this fashion.[8] Similarly, the delayed processing scheduled by the **`timeout()`** function is done by a kernel thread, `callout_thread`, running at the highest *sys* class priority. In the default configuration, this thread runs at a priority immediately below the realtime class priorities. As before, at every clock tick, the `callout_thread` checks to see if any timeout processing needs to be done. The difference is that, instead of being called at interrupt level, now it simply sleeps on a condition variable periodically signalled by the clock interrupt. The `callout_thread` runs only after all the realtime threads run, thereby avoiding priority inversion due to delayed processing.

8. Eykholt 1992.

Yet realtime threads may need to set a timer/alarm so that they can be awakened in realtime. This problem has been addressed by adding a function called **realtime_timeout()**. The requests made via **realtime_timeout()** are run at the lowest interrupt level and are kept on a separate heap. Figure 2-4 shows the priority space model before and after the changes for **realtime_timeout()**. Only time-critical interfaces use this function. For example, the interval timer and its interface **setitimer()** are based on **realtime_timeout()**. The interfaces for **timeout()** and **realtime_timeout()** are identical as shown in Table 2-2.

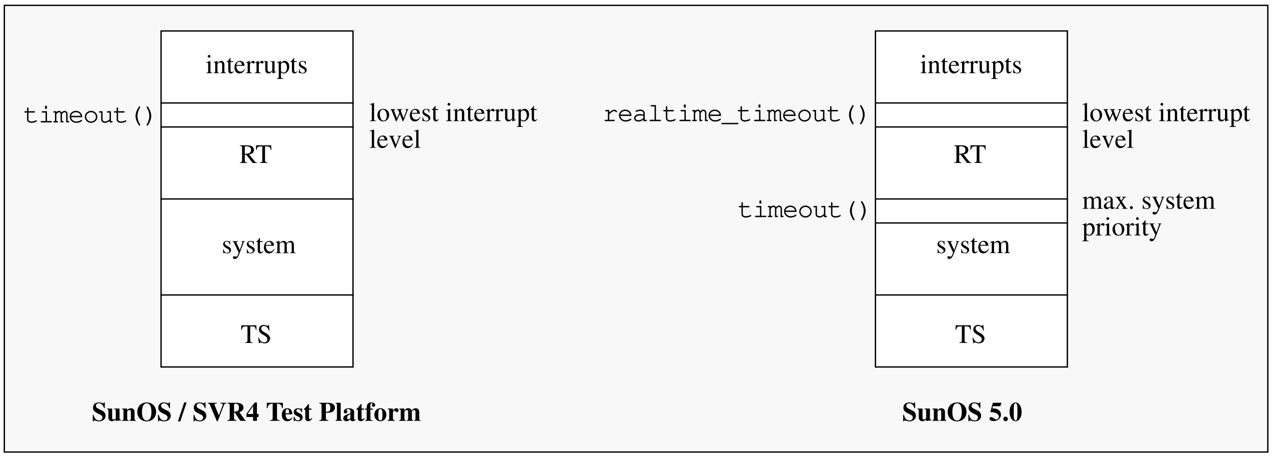


Figure 2-4 Callout-Queue Processing

The Priority Inversion Problem

The priority inversion problem was first identified by Lampson and Redell in their discussion on the use of monitors in Mesa.[9] It has received a moderate amount of attention lately in the literature.[10] [11] What follows is a brief description of the problem as it pertains to SunOS. SunOS 5.0 is a multithreading kernel that uses synchronization objects such as *mutexes* and *readers/writer locks* to enforce thread synchronization. The use of such

9. B. W. Lampson and D. D. Redell, "Experiences with processes and monitors in Mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 105-117, February 1980.

10. Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," *Proceedings of the Real-Time Systems Symposium*, December 6-8, 1988, Huntsville, AL [hereafter referred to as "Rajkumar 1988"].

11. Lui Sha, Ragunathan Rajkumar, John P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990 [hereafter referred to as "Sha 1990"].

synchronization objects can lead to uncontrolled priority inversion. By way of illustration, consider the case where a high priority thread that uses a mutex gets blocked during the periods of time when a lower-priority thread that owns the mutex is preempted by intermediate-priority threads. These periods of time are potentially very long—in fact, they are unbounded, since the amount of time a high priority thread must wait for a mutex to become unlocked may depend not only on the duration of some critical sections, but on the duration of the complete execution of some threads. Figure 2-5 below illustrates these situations. Case A shows simple priority inversion: low priority thread T_2 blocks high priority thread T_1 because T_2 holds synchronization object S_0; Case B is identical, save that medium priority thread T_3 preempts the execution of T_2, with the result that T_1's blocking time depends on the duration of T_3's execution.

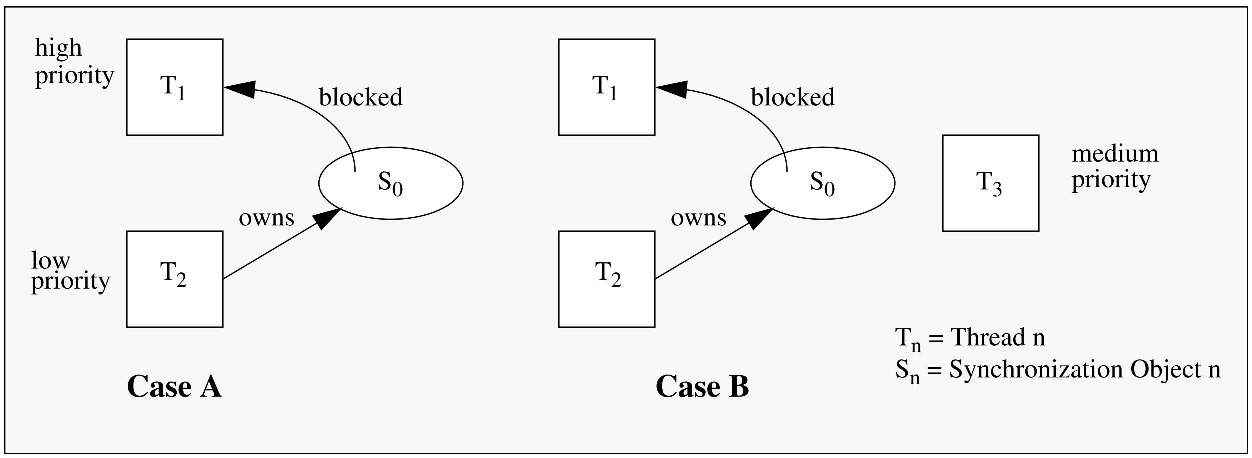


Figure 2-5 Bounded and Unbounded Priority Inversion

To solve the priority inversion problem, we have chosen to implement the *basic priority inheritance protocol*. A complete and detailed discussion of this protocol is beyond the scope of this paper, but its essentials are easily described. A more elaborate description may be found in Sha.[12] The basic priority inheritance protocol attempts to limit the duration of priority inversion during blocking by having a blocked high priority thread propagate (will) its priority to all lower priority threads that block it. When lower priority threads cease to block a high priority thread, the lower priority threads revert to their original priority.

12. Sha 1990.

The basic priority inheritance protocol concerns itself with two general abstract data types. These abstract data types are the *schedulable entity* (in our case, the thread) and *synchronization objects* (in our implementation, mutexes and readers/writer locks).

Let us consider what happens in the general case when threads acquire and release synchronization objects. Before a thread T_1 enters a critical section, it attempts to acquire ownership of the synchronization object S guarding the critical section. If the synchronization object S is already owned by thread T_2, the attempt by thread T_1 to acquire synchronization object S fails, and thread T_1 blocks. In this scenario, thread T_1 is said to be blocked on synchronization object S and blocked by thread T_2. If the synchronization object S is not already owned by another thread, thread T_1 will acquire ownership of synchronization object S and enter the critical section that it guards. When thread T_1 exits this critical section, it releases synchronization object S and awakens the highest priority thread blocked on S by T_1.

Priority inheritance determines at what priority threads block and are dispatched. Priority inheritance makes a distinction between the global priority and the *inherited priority* of a thread. The inherited priority is the priority a thread obtains via priority inheritance by blocking higher priority threads. The *dispatch priority* is computed as the maximum of the global and inherited priorities of a thread. A thread T executes at its global priority unless it is in a critical section and is blocking higher priority threads. If thread T blocks higher priority threads, T inherits a priority equal to the maximum dispatch priority of the threads it blocks. When thread T exits a critical section and releases the associated synchronization object, it relinquishes the inheritance it obtained by holding the synchronization object.

To ensure that the duration of priority inversion is bounded, the basic priority inheritance protocol requires that priority inheritance be transitive. That is, if there are three threads, T_1, T_2, and T_3, such that the dispatch priorities are ordered thus:

$$\text{Priority}(T_1) > \text{Priority}(T_2) > \text{Priority}(T_3)$$

and if thread T_3 blocks thread T_2, and thread T_2 blocks thread T_1, then thread T_3 inherits the priority of thread T_1 via thread T_2. Figure 2-6 below illustrates a case where transitive priority inheritance prevents unbounded blocking by the highest priority thread: if runnable thread T_3 does not inherit indirectly from blocked thread T_1, then the medium priority runnable thread T_4 will preempt the execution of T_3, resulting in unbounded blocking for T_1.

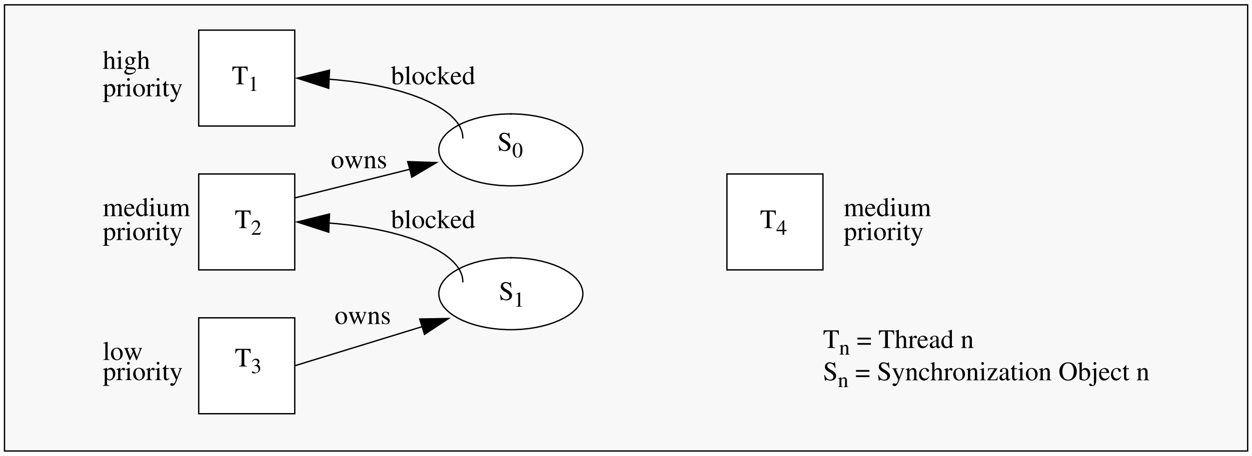


Figure 2-6 Indirect Blocking and Transitive Priority Inheritance

As described in the literature,[13] the basic priority inheritance protocol imposes bounds on the duration of blocking. Briefly, if there are *m* lower priority threads, and these threads access *k* distinct synchronization objects in common with a higher priority thread, the higher priority thread can be blocked by at most a number of critical sections equal to the minimum of *m* and *k* .

Priority Inheritance Primitives

Table 2-3 summarizes the operations of priority inheritance. Priority inheritance for both mutexes and readers/writer locks is implemented using these primitives. The **pi_willto()** function is used when a thread attempts to acquire a synchronization object and discovers that it is blocked by another thread. The **pi_waive()** function is used when a thread releases ownership of a synchronization object and must surrender the inheritance that it received via the synchronization object. These two functions are the primary operations in our priority inheritance implementations. Note that the function **pi_willto()** contains no direct reference to the synchronization object that the argument thread is blocked on. This is because **pi_willto()** is called only after the argument thread has been put on the sleep queue for the

13. Rajkumar 1988.

synchronization object. Doing so causes information to be saved within the thread structure itself, allowing the synchronization object to be found via the thread.

Table 2-3 Priority Inheritance Primitives

Function Name	Operation
void pi_willto (*thread*)	Will the priority of the argument thread to all threads directly or indirectly blocking it.
void pi_waive (*thread, sync_object*)	Release the priority inheritance that the argument thread obtained via a particular synchronization object.

Priority Inheritance Implementation

Priority inheritance comes into play only when threads block on a synchronization object. To implement transitive priority inheritance, when starting from an arbitrary synchronization object, priority inheritance must be able to find the owning thread and each successive synchronization object and thread in the blocking chain. In order to make this information quick and easy to obtain, each synchronization object maintains a pointer to its owning thread; likewise, each blocked thread keeps a pointer to the synchronization object it is blocked on and a tag field to identify the synchronization object's type. Using this information, the priority inheritance mechanism in SunOS 5.0 can follow a blocking chain to the end. The end of the blocking chain, as far as priority inheritance is concerned, arrives when it finds a thread that is not blocked, or a synchronization object that is not priority inverted.

The distinction that priority inheritance makes between global priority and inherited priority is reflected in the implementation. The thread structure contains a field for each. Ordinarily, the field that represents the thread's inherited priority is set to zero. If the thread inherits a priority, the thread is marked to indicate this condition, and the inherited priority field is set accordingly. The code that enqueues and dequeues threads makes note of whether the thread possesses an inherited priority and uses it instead of the global priority when appropriate. Related to this issue, when priority inheritance encounters a thread that is in a sleep queue or dispatch queue, it must first dequeue the thread; will the new, inherited priority to the thread; and then re-enqueue the thread at its new priority. Currently, in order to maintain the consistency of global data structures, all these operations must take place while holding `schedlock`, the global scheduler lock (see earlier section on “Scheduling”).

Note further that it is possible for a thread to hold several synchronization objects at once, each of which could be priority inverted at potentially different priorities. When priority inheritance wills a priority to a priority inverting thread, it saves a record of the synchronization object and the priority of the highest priority thread blocked on it in a circular linked-list in the thread structure of the priority inverting thread. When the priority inverting thread releases a synchronization object, the record of its inheritance from this synchronization object is removed from the thread's linked-list, and the linked-list is traversed to compute the thread's (possibly new) dispatch priority.

The sleep queue and all the priority inheritance information associated with a synchronization object are encapsulated in an abstract data type called a *turnstile*. Figure 2-7 below displays this relationship. Because there are many, many synchronization objects in the system, we allocate turnstiles dynamically when needed. When a thread blocks on a synchronization object that previously had no threads blocked on it, the newly blocked thread allocates a turnstile from a pool of turnstiles and attaches the turnstile to the synchronization object. The pool itself grows with the number of allocated threads in the kernel. The existence of a turnstile attached to a synchronization object is an indication that there are threads blocked on a synchronization object. When a thread releases a synchronization object that it owns and in so doing discovers that it has awakened the last of the threads in the turnstile's sleep queue, the releasing thread returns the turnstile to the free pool.

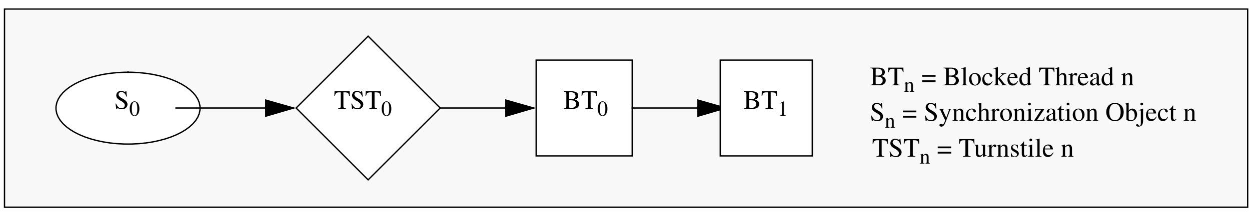


Figure 2-7 Turnstiles

The traditional UNIX implementation of sleep queues uses a hashing scheme based on a *wait channel*, which is usually the address of a desired resource or some offset therefrom; aliasing of hash buckets is implicit in such a scheme.[14] High priority processes sleeping on unrelated wait channels may hash to the same sleep queue bucket; thus to wake up processes sleeping on a particular resource, it is necessary to hash to the appropriate sleep queue bucket, and

14. Leffler 1989.

then traverse the sleep queue looking for those processes with matching wait channels. The result is that the behavior of insertion and release operations on sleep queues is not bounded by the number of processes competing for the resource—instead, the bound is the number of processes in the system.

Rather than using hashing schemes, turnstiles provide a per-synchronization object sleep queue, so aliasing is avoided. Insertion and release operations on turnstiles are bounded only by the number of threads competing for the associated synchronization object. Since threads waiting in a turnstile are queued in priority order, higher priority threads have a fixed bound for behavior. Turnstiles have potential to improve the performance of sleep and wakeup operations on large systems with lots of threads.

Priority Inheritance for Readers/Writer Locks

The case of readers/writer locks deviates somewhat from the idealized picture we have presented for synchronization objects. When a synchronization object of this type is acquired by a thread with the intention of updating the data it protects (i.e., when a writer lock is in effect), there is no deviation: there is a single thread that controls the synchronization object and is the beneficiary of all inheritance applied via the synchronization object.

The deviation comes in the case of readers locks. Readers locks can have a potentially large number of "owners." It is not practical in terms of space to keep a pointer in the readers lock to every thread that currently "owns" it. To simplify the implementation, for readers locks we have implemented what we term the *owner-of-record*. When a readers/writer lock is acquired for reading, the first thread that obtains the readers lock is assigned "ownership" of the synchronization object. As such, it is the beneficiary of all inheritance that passes through the readers lock until it releases the lock. When the owner-of-record thread releases a readers lock, it is possible that there are still other threads that own it. These owners are in a sense anonymous, since they cannot be identified by inspecting the readers lock, nor can they inherit from it while they own it. Since it is not an uncommon condition for a readers lock to have a single owner, our measurements indicate that there is still value in providing this limited form of priority inheritance for readers locks.

Lessons Learned

In the implementation of the realtime scheduling features, we encountered two problems: the requirements on mutex implementation and the generalization of priority inheritance.

Mutex Entry/Exit Implementation

Priority inheritance requires the owner of a synchronization object to be known. Our basic mutex entry code appears as follows:

```
ldstub    [MUTEX_PTR + M_LOCK], LOCK_REG
tst       LOCK_REG
bnz       lock_already_held
nop
st        OWNER_ID_REG, [MUTEX_PTR + M_OWNER]
```

An interrupt occurring between the `ldstub` and `st` instructions can leave the mutex held by an unknown owner.

One possible solution is to raise the processor interrupt level to prevent interruption during the above sequence. Since the management of the processor interrupt level is non-trivial, efficiency requires that uncontested entry/exit should not require raising the processor interrupt level. Another solution might be to use an atomic operation which could set the lock and owner fields simultaneously, but in the SPARC architecture,[15] the sole atomic update available is the `ldstub` instruction. A third solution might be to have a blocking thread arrange for deferred inheritance, which the acquiring thread would have to check for and assume after return from interrupt. Again, this solution was rejected for its performance cost.

We chose to solve this issue by constraining the mutex lock operation to a stylized behavior which could be recognized by the interrupt handler. The use of a fixed register as `LOCK_REG`, with the convention that a zero value in this register indicated the successful acquisition of a mutex, allows the interrupt handler to safely set the mutex owner before processing the interrupt. Not only does this work in the uniprocessor case, where no other thread ever sees the lock held with an unknown owner, but it works in the multiprocessor case because the interval between setting the lock and setting the owner is bounded, and a thread attempting to acquire the mutex from another processor can spin until the owner is known.

---

15. *The SPARC Architecture Manual, Version 8*, Prentice-Hall 1991.

In order to prevent erroneous inheritance, possibly to a thread that is no longer extant, we chose to invalidate the owner field of a mutex at exit. This invalidation could not take place after the lock was released due to the possibility of a race with a thread acquiring the mutex from another processor. It could not take place before the lock was released, due to the possibility of taking an interrupt between clearing the owner and the lock, and being unable to determine unambiguously whether the lock was held by the interrupted thread or some other. Hence we were forced to encode the owner and lock fields within the same word, and clear both with a single instruction.

Limitations in Providing Priority Inheritance

Priority inheritance is appropriate only when it is known which thread or threads must release the synchronization object in order for the blocking thread to proceed. We faced this issue for readers/writer locks, due to the cost of maintaining the list of reader access holders.

For condition variables and semaphores, priority inheritance is not possible. The protocol for these synchronization objects does not determine which thread(s) will release these objects. This is particularly unfortunate for condition variables, which can be used with mutexes to build higher order synchronization operations. Even if the protocol for these synchronization objects determines which threads(s) will be releasing them, the inability to inherit through condition variables precludes the provision of inheritance.

Performance Measurement

Scheduling performance from the realtime perspective can be defined in terms of the *dispatch latency*. We define dispatch latency as the amount of time it takes to begin execution of a high priority runnable thread. As illustrated in Figure 2-8, this time includes the remainder of the non-preemption interval for the currently running thread, the time to resolve resource conflicts represented by acquiring held synchronization objects, and the context switch time. We do not include the processing time associated with interrupts nor any of the time within the application used in responding to the event which made the thread runnable. The total application response time will of course include both interrupt and application processing time.

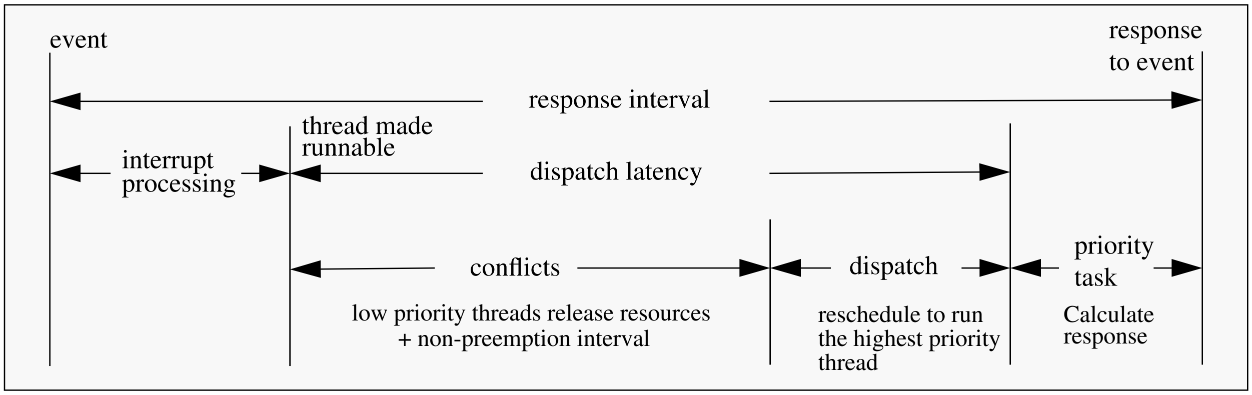


Figure 2-8 Dispatch Latency

We examined two different ways of measuring dispatch latency. The first way is to measure all possible paths in the kernel to see if there exists a path which exceeds the response time we wish to guarantee. This requires a complete static analysis of the kernel, and the use of dynamically loaded modules again makes this solution infeasible. The second way is to repeatedly sample the dispatch latency of the kernel under specified loads. With a sufficiently large number of samples, this second way should have the same result. To measure the dispatch latency of SunOS 5.0, we have chosen the sampling approach.

Our sampling dispatch latency test (SDLT) is composed of a sampling process, using a sample driver and recording violations for further analysis. The sampling driver provides two **ioctl()** commands to control sampling. The first command computes and records the time for a wakeup event, blocks the sampling process, and scheduling the wakeup via **realtime_timeout()**. When the wakeup callout function is called, it unblocks the sampling process, enables kernel execution path recording, and records the time of kernel wakeup. When the thread resumes execution, it records the time of kernel dispatch, and the first **ioctl()** then returns the recorded times to the sampling process. The sampling process then uses the second **ioctl()** command to disable kernel execution path recording and attain a timestamp for the return to user mode. The execution path recording has been extended to include timestamps to enable detailed analysis. When the sampling process detects a dispatch latency violating our projected guaranteed value, it captures in a file the violation time parameters and execution path trace.

We have measured our kernel with preemption both disabled and enabled. With preemption disabled, we have observed dispatch latencies in excess of 100 milliseconds. Our initial results with preemption enabled showed a dispatch latency of about 2 milliseconds on the SPARCstation 1, with a small number of larger values. Our analysis of these larger values pointed to long non-preemption intervals associated with the memory management unit hardware layer.[16] Subsequent work reduced these intervals, and the following table indicates some measured values for dispatch latency.[17]

Table 2-4 Realtime System Dispatch Latency with SunOS 5.0

Sun Workstations	**Dispatch Latency**
SPARCstation 1	4.5 milliseconds[1]
SPARCstation 1+	4.0 milliseconds[1]
SPARCstation IPX	2.2 milliseconds[2]
SPARCstation 2	2.0 milliseconds[3]

1. The worst case values are incurred when the MMU context is stolen. In a system with less than eight active processes, the dispatch latency is less than 2.0 milliseconds.
2. The worst case values are incurred when the MMU context is stolen. In a system with less than eight active processes, the dispatch latency is less than 1.0 milliseconds.
3. The worst case values are incurred when the MMU context is stolen. In a system with less than sixteen active processes, the dispatch latency is less than 1.0 milliseconds.

Note the effects that can arise when the context of the realtime process is stolen by another process. The operations associated with reloading the realtime process's context can take as much as 4 milliseconds on a SPARCstation 1, much of which is non-preemptible. We discuss a possible solution to this problem in the next section on "Future Work."

Future Work

Although we feel that we have achieved much in bounding the dispatch latency of SunOS, there remain a number of areas where we could improve the performance of the system, or increase its utility as a base for realtime applications. In this section, we discuss some of the areas we are interested in improving.

---

16. Joseph P. Moran, "SunOS Virtual Memory Implementation," *Proceedings for the Spring 1988 EUUG Conference*, EUUG, London, England, Spring 1988.

17. Excerpted from *SunOS 5.0 Realtime Programmer's and Administrator's Guide*, part number 800-7377-10.

- **Realtime I/O**

In SunOS 5.0, much of the I/O processing is done through STREAMS. The STREAMS processing is done at the systems priority level and thus executes below any active realtime thread. It is impossible to guarantee realtime I/O for STREAMS without drastically changing the handling of STREAMS processing. Other I/O activities require additional changes in request queuing to gain realtime priority-based behavior.

- **Dispatch Latency**

We intend to continue trying to reduce the dispatch latency of the system. Work is going on to improve the granularity of the locking and to reduce the length of non-preemption intervals.

While we have been able to bound dispatch latency, applications are interested in the overall response time. One of the components of response time is the time required to process interrupts. Arrival of multiple interrupts may delay dispatch, thus making the duration of interrupt processing unbounded. We intend to provide the user a mechanism to block certain interrupts temporarily when running in a multiprocessor environment. Such a capability provides a means of shielding realtime threads from the delays due to interrupts.

- **Locking Contexts**

Since realtime threads are often event-driven, they typically do not run very often. On machines such as the SPARCstation 1 that implement the older, large kernel-managed TLB-based MMU, it is possible for a realtime thread to have its context stolen, even though its pages may be locked into primary memory. In order to resume the realtime thread, it may be necessary to steal a context from another thread and reload the realtime thread's context. To avoid the long non-preemption points associated with this activity, we are contemplating implementing a facility to lock a context into association with a thread, providing a facility analogous to that for locking pages into memory.

- **POSIX 1003.4**

The POSIX P1003.4 Realtime Extension to POSIX 1003.1 is nearing approval as a standard. This amendment will provide queued signals with data, asynchronous and synchronized I/O interfaces, counting semaphores, memory locking and shared memory, execution scheduling, message passing, and realtime files. We are working on providing compliance for this eventual standard.

Summary

In SunOS 5.0, we have provided the following realtime functionality:

- Static priority and fixed quantum scheduling for realtime threads.
- A fully preemptive kernel, providing bounded dispatch latency.
- Hidden scheduling in the kernel has been greatly reduced, eliminating much priority inversion.
- Priority inversion arising from the use of synchronization objects has been controlled by implementing the basic priority inheritance protocol.

Acknowledgments

The authors are indebted to the multithreading group of the Operating Systems Technology Group of SunSoft, both for providing the platform multithread implementation and for their assistance in development of the realtime scheduling mechanisms. We also wish to acknowledge the assistance of Deepak Dhamdhere for his contributions to the Performance section of this paper. He built many of the measurement tools and collected the data that provided a basis for that section.

Biographies

Sandeep Khanna is a software engineer in the Operating Systems Technology Group at SunSoft. He graduated with an M.S. in Computer Science from the University of Mississippi in 1987. He received his B.E. in Electrical and Electronics Engineering from Birla Institute of Technology and Science, Pilani, India in 1984. He is currently involved with the design and implementation of realtime extensions to SunOS 5.0.

Michael Sebrée is a software engineer in the Operating Systems Technology Group at SunSoft. He is rumored to have obtained some knowledge of operating systems in general, and the UNIX operating system in particular, at Vanderbilt University and at UCLA.

John Zolnowsky is a software engineer in the Operating Systems Technology Group at SunSoft. He has a Ph.D. in Computer Science from Stanford University, was instruction set architect for the 68000 processor family, and now serves as technical editor for the POSIX 1003.4 Realtime Extensions.

Languages 3

The following articles are drawn from the Sun document *Introduction to SPARCworks* (part number 800-7262-11).

SPARCworks and the Software Development Cycle

As every programmer knows, the task of creating high-performance, bug-free software has become more difficult as applications grow larger and more complex. Innovations such as object-based design and object-oriented languages have helped somewhat to manage the demands of increasing complexity. In addition, programmers now insist on better tools to help them meet the ever-present pressures of schedule and budget. SPARCworks is a set of six such programming tools that exploits the graphics capabilities of the OpenWindows windowing system and its interprogram communications capabilities. The tools are:

- **SPARCworks Manager** — For managing and coordinating other SPARCworks programming tools
- **SourceBrowser** — For statically analyzing source code and graphing program structure
- **Debugger** — For dynamically analyzing programs, observing behavior while the program runs, and collecting information for performance profiling
- **Analyzer** — For tuning program performance, including memory allocation
- **FileMerge** — For merging source files and coordinating source code changes
- **MakeTool** — For building programs and browsing makefiles

SPARCworks tools form part of an overall development environment that includes the standard operating system programming utilities, the DeskSet productivity tools (including File Manager and Mail Tool), and the OpenWindows Developer's Guide for interactively building OpenWindows graphical interfaces.

SPARCworks tools simplify the tasks programmers perform most often: coding, compiling, debugging, and performance-tuning programs. Each tool is designed to fill a specific need and at the same time to cooperate with other SPARCworks tools. The result is a flexible, integrated toolset that solves the most important problems faced by the individual programmer.

The remainder of this article explains which SPARCworks tools are useful at each stage of the software development process.

Conceptualizing

The early steps in any software design project are conceptual. The first step is to clearly define the problems that the software is meant to solve; the second is to propose solutions to these problems. The proposed solutions give rise to a specification, set of core algorithms, and an overall design approach.

For procedural programming, the result of the conceptualization stage is a detailed set of tasks that the program must perform, divided into functions that will implement the program's algorithms. For object-based programming, the result is a hierarchy of classes, a listing of the responsibilities of each class, and the ways objects of each class will collaborate with other objects to fulfill their responsibilities.

This early stage of software design is arguably the most critical to the long-term success of a project. Regrettably, it is also the stage that is least amenable to computer-aided assistance, relying as it does on a clear assessment of the problem by the designers and an analysis of trade-offs between desired features and available resources.

Because SPARCworks concentrates on helping the individual programmer to write efficient code quickly, it leaves assistance in such tasks as high-level program design, large team development efforts, and revision control to other computer-aided software engineering (CASE) tools.

Coding

Because each developer has a favorite editor, SPARCworks introduces no new editor of its own. If you choose to use the SPARCworks Manager, you can place the editor of your choice in iconic form in the Manager window so that you can easily start it the way you do other SPARCworks tools. In some cases, you can associate an instance of the editor to a particular SPARCworks session.

After the compile-debug loop has begun, SPARCworks debugging tools provide editing windows so that you can make changes to source code quickly within the tool, recompile, and continue debugging.

Building Executables and Libraries

Most everyday compilation and linking on Sun systems is done with `make`, the utility that ensures that programs are built from the newest sources according to lists of rules contained in makefiles. While the value of `make` in maintaining large projects is widely recognized, writing and maintaining a makefile has always been an annoying penalty to pay for its use.

SPARCworks simplifies the use of `make` and makefile maintenance with MakeTool, a tool that expands makefile rules and macros so that they can be easily understood. In addition, MakeTool automatically creates a menu of high-level targets whenever it loads a makefile. You can use the menu to build the targets. MakeTool updates the menu automatically whenever you enter options and arguments to the `make` command.

SPARCworks analysis tools (Debugger, SourceBrowser, and Analyzer) require that programs be compiled with special flags before the tools can be used to their full capabilities. These flags create special databases or symbol tables for the tools, and only compatible compilers can respond properly to these flags. The following SPARCompilers can be used with SPARCworks tools:

- SPARCompiler C / ANSI C
- SPARCompiler C++
- SPARCompiler FORTRAN
- SPARCompiler Pascal

Other Sun compilers, such as Sun Ada, provide their own programming environments and can only interact in a limited way with SPARCworks analysis tools.

Debugging

After a program successfully compiles, the search for errors (bugs) begins. Two SPARCworks tools assist in the debugging effort:

- Debugger, which examines the values of variables and address locations while you step through the program.
- SourceBrowser, which enables you to find any or all instances of a specific function or variable in source code. SourceBrowser is especially useful for analyzing unfamiliar code that you are required to maintain.

These two tools, one a dynamic analyzer and the other a static analyzer, form the heart of the SPARCworks toolset. Used together, they help you quickly find the source of a bug and correct all its instances. The capabilities of these two tools are extensive, making their features difficult to summarize. Their use is treated in detail in *Introduction to SPARCworks*, both separately and as a team.

Merging Source Code

Source files have a way of proliferating, even when only one programmer is working on a project. When several versions of a source file must be merged, programmers have until now relied on the `diff(1)` operating system utility and their favorite text editor, but the development of sophisticated window interfaces has made the `diff` command-line interface obsolete.

SPARCworks FileMerge provides a convenient way to merge source files or entire directories of source files quickly. You can load two files into FileMerge and merge them, or you can specify a third file, called the *ancestor* of the two files (which are called its *descendants*). When you specify an ancestor file, FileMerge marks lines in the descendants that are different from the ancestor and automatically produces a merged file based on all three files.

Testing and Performance Tuning

After you have successfully compiled a program and eliminated its major bugs, you will want to evaluate its performance. The SPARCworks Analyzer measures and displays a program's performance profile, suggesting ways to improve performance. The Analyzer effectively supersedes operating system utilities such as `prof(1)` and `gprof(1)`, providing capabilities not offered by any existing standard utility.

The Debugger collects performance data for the Analyzer and places the data in a file. The Analyzer then examines the collected data and presents its analysis in a variety of graphical and text displays. Because data collection relies on operating system calls, not on Debugger symbol tables, any program can be evaluated with Analyzer — no special compiler options are required.

Note – Performance data can be collected only when Debugger is running under SunOS 5.0 (Solaris 2.0).

Maintaining Software

For every programmer who initially codes a program, several others inherit the task of maintaining it. Programmers who inherit code find SPARCworks SourceBrowser an indispensable tool for grasping the structure of an unfamiliar program. SourceBrowser's ClassGrapher and CallGrapher provide detailed overviews of a program's organization by graphing its class inheritance and function call trees.

Inherited bugs are difficult to fix, not only because inherited code is less familiar than code that you authored but because inherited bugs are recognized late in the design cycle and are usually more subtle than bugs recognized earlier. The combination of SourceBrowser and Debugger form a powerful team for tracking down elusive bugs. For example:

- With SourceBrowser, you can browse source code and quickly find every instance of a particular variable, function, or object. The tool contains a ClassBrowser that performs the same functions for class attributes in C++ programs. If you find a questionable line of source code, you can move to Debugger, set a breakpoint at that line, and examine the values taken on by a variable during execution.
- After you stop at a questionable function in Debugger, you can go to SourceBrowser to find how the function is defined. You can also find all the other places where it is used.

System Requirements

To run SPARCworks effectively, your system should have at least 16 Mbytes of memory and 40 Mbytes of free disk space.

SPARCworks Manager

SPARCworks Manager is a graphic interface for coordinating SPARCworks tools and controlling the programming environment. Figure 3-1 shows the SPARCworks Manager window with SPARCworks tool icons displayed.

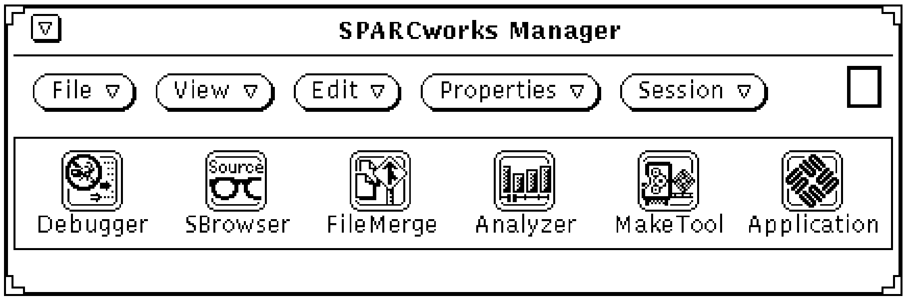


Figure 3-1 SPARCworks Manager

Each tool can be started by double-clicking on its icon or by dragging the icon onto the screen workspace. A tool that has been started from the SPARCworks Manager is associated with that instance of SPARCworks Manager, and can be controlled as part of a management *session*.

Besides managing the other five standard SPARCworks tools, SPARCworks Manager can also control custom applications written by users or third-party developers. When integrated with SPARCworks and started with SPARCworks Manager, the standard SPARCworks tools and custom applications combine to provide you with the following advantages:

- **Drag-and-drop startup and load** — You can drag a file name or DeskSet icon and drop it on a SPARCworks tool, which recognizes whether the object can be loaded or otherwise acted upon.
- **Session control** — You can group development activities into logical management *sessions*. Multiple sessions can proceed simultaneously, each controlled by its own SPARCworks Manager.
- **Workspace organization** — SPARCworks tools associated with a management session can be closed to a single icon.
- **Central control** — The SPARCworks Manager can set environment variables and the working directory for all SPARCworks tools used in a session.

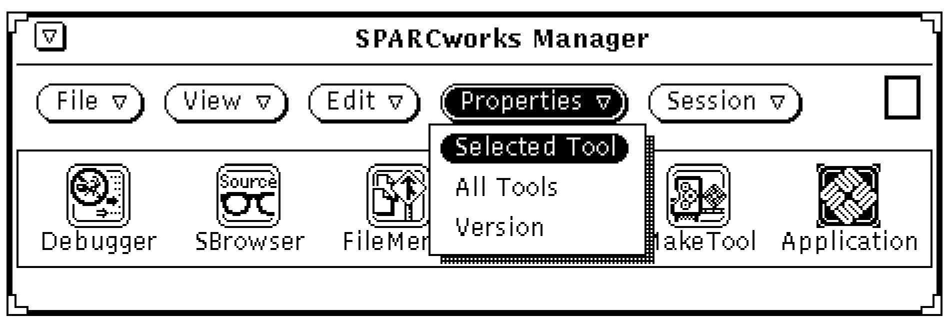
Extending the SPARCworks Manager Palette

SPARCworks Manager gives you control of the commands issued when you double-click on a tool icon or drag it into the workspace, which means that the icon for any tool on the palette can be customized and used to start any application.

One tool that most developers find convenient to place in the SPARCworks Manager palette is the application under development. After your application has been successfully compiled, you can set properties to run the application from the SPARCworks Manager. Here's how:

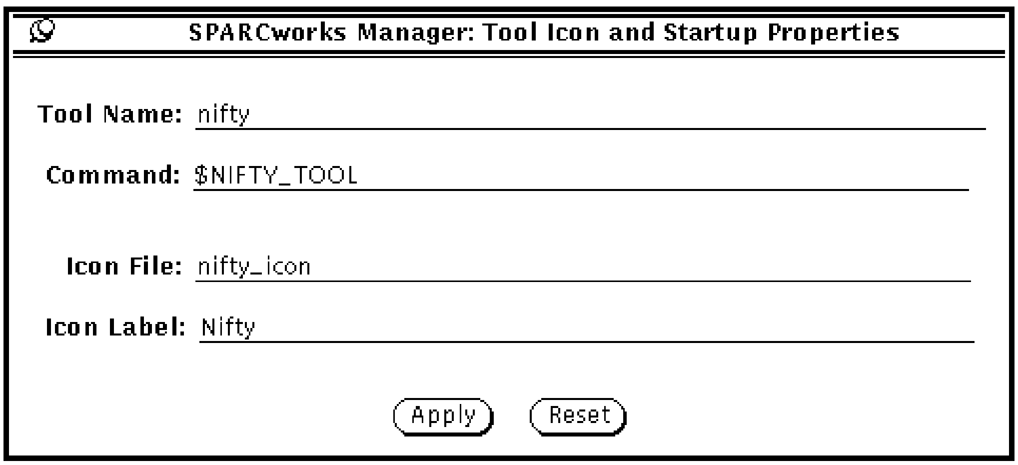
1. **Select the Application icon (or other icon) and choose Duplicate Tool from the Edit menu.**

2. **Select the icon (or its duplicate) again and choose Selected Tool from the Properties menu.**
An unaltered copy of the icon should be left available to start the tool for which it was intended, or, in the case of the Application icon, for use with the drop target (as described in the next section, "Drag-and-Drop Startup").

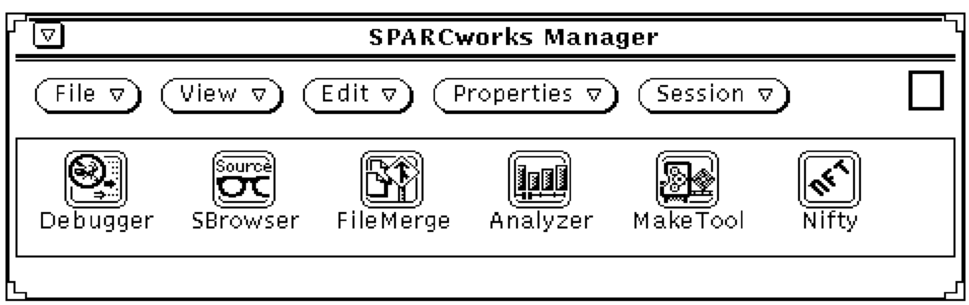


3. **Edit the command line in the pop-up window that appears so the SPARCworks Manager will start up your application when the icon is dragged onto the desktop.**
   If desired, change the icon label (the text that appears underneath the icon when it is displayed on the palette), the tool name, and icon file name (which directs SPARCworks Manager to a file that contains a custom icon).

4. **Apply your changes by clicking on the Apply button.**
   The figure shows an environment variable, `NIFTY_TOOL`, being used for the command. This technique allows flexibility. If you use it, be sure that you set the variable to the path name of the command that starts the tool.



After you apply your changes, the Application icon and its label represent the new tool. The following figure shows the SPARCworks Manager palette that results from applying the changes for NiftyTool — the Nifty icon (stored in the file `nifty_icon`) is shown on the right, replacing the Application icon.



By duplicating any tool on the palette and customizing the properties of the copy to execute a new command (and display a new icon), you can easily include any tool of your choice.

The most common tool to integrate is a text editor, because so many developers prefer their favorite editors. Figure 3-2 shows a tool properties window set up for starting the `vi` text editor in a Command Tool window. The `$SUNPRO_SWM_GUI_ARGS` environment variable specifies the initial position of windows when drag-and-drop is used. See *Managing SPARCworks Tools* (part number 800-6622-11) for definitions of environment variables used by SPARCworks Manager.

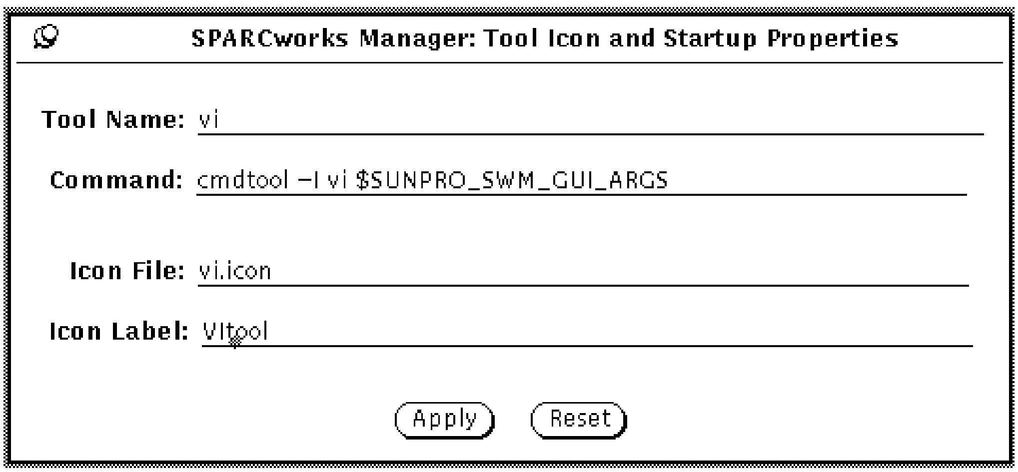
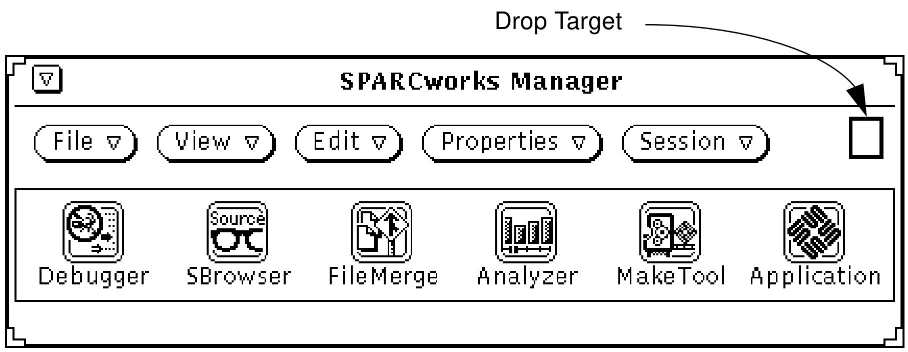


Figure 3-2 Setting Up a Tool for the `vi` Editor

Drag-and-Drop Startup

An OpenWindows drop target is located in the upper right corner of the SPARCworks Manager base window. The drop target is a convenient way to specify an arbitrary application and start it from SPARCworks Manager.



To start a non-SPARCworks application from the SPARCworks Manager palette:

1. **Select the application's File Manager icon or full path name.**
2. **Drag the icon or path name (with the SELECT mouse button) over the SPARCworks Manager drop target.**
3. **"Drop" the name or icon by releasing the mouse button.**
   This step identifies the application and associates it with the Application icon on the SPARCworks Manager palette.
4. **Start the application by double-clicking on the Application icon or by dragging it into the workspace.**

Because session control only works with tools that communicate using the ToolTalk inter-process communication protocols, not all applications you start this way will respond to session control commands.

Session Control

An important capability provided by the SPARCworks Manager is *session control*.

Session control means that all the SPARCworks tools you start from an instance of SPARCworks Manager are part of a single session, and can be opened, closed, or hidden with one SPARCworks Manager command.

Session control applies to any custom tools you have placed on the palette yourself, provided the tools understand the protocols of the ToolTalk software. This feature helps eliminate the confusion that results from having several instances of a tool active on screen at the same time and greatly reduces on-screen clutter.

You can also set properties that are common to all the session tools from the SPARCworks Manager: the working directory in which new tools will be started and the startup environment variables for the tools. Figure 3-3 shows the Properties window that appears when you choose All Tools from the Properties menu. For details on the environment variables shown in the window, see *Managing SPARCworks Tools* and the operating system documentation for the `sh` and `setenv` commands.

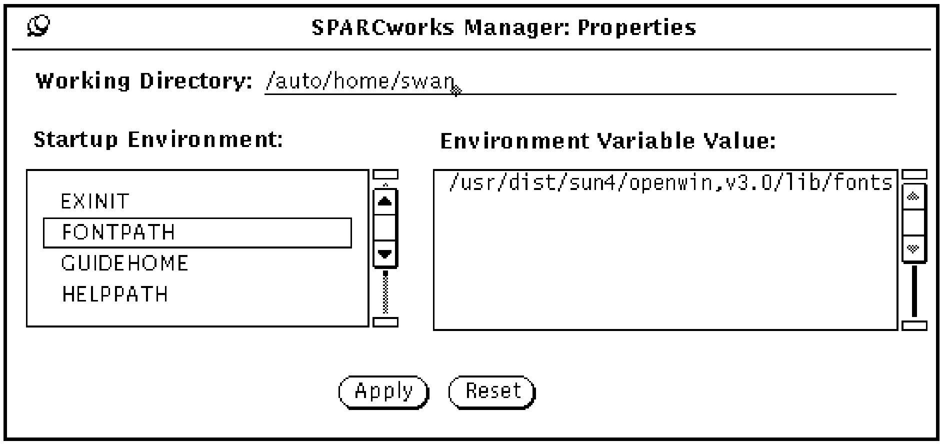


Figure 3-3 SPARCworks Manager Properties Window for All Tools

SourceBrowser

A programmer can use one of two perspectives to gain insight to a piece of software:

- **Dynamic analysis**, which involves observing the run-time behavior of a program, including such factors as how many times each function is called in typical usage, how the program uses memory, and how the value of an observed variable changes during execution. SPARCworks Debugger and Analyzer are tools for dynamic analysis.
- **Static analysis**, which involves observing the structure of a program as defined by its source code, including the identity of called and calling functions, graphs of class genealogy, and the organization of header files and defining statements. SPARCworks SourceBrowser is a tool — actually, a family of tools — for static analysis.

SourceBrowser as Static Analyzer

Static analysis of a compiled program primarily means examining its source code in an organized fashion. The program does not need to be executing during static analysis. Searching through high-level source code with a text editor is a primitive form of static analysis.

SourceBrowser performs queries on source code (analogous to searching in a text editor) that are constrained by the semantics of the source language it is browsing and by other, user-imposed, constraints. In addition, SourceBrowser uses an auxiliary database (created by the `-sb` SPARCompiler command-line option) to speed searches through program source. By creating a separate database rather than embedding its information in the executable program, SourceBrowser does not add to the size of the compiled program.

SPARCompilers inform SourceBrowser of the source files that were compiled to produce an executable or library so that they can be loaded and browsed automatically. By default, SourceBrowser queries all source files, but you can limit the query to focus on any subset of source files. SourceBrowser considers all header files to be source files during queries. The SourceBrowser database is discussed in more detail in the later section, “SourceBrowser Data Flow Overview”.

Graph and Browser Windows

SourceBrowser is the base application for launching several interactive windows that give you alternate views of source:

- **SourceBrowser base window** — Opens when you start SourceBrowser. It displays source code, performs queries, and provides access to the other major windows.
- **CallGrapher** — Helps you visualize and understand function relationships in your program. It displays a function call tree to identify the functions that either call or are called by one or more functions you select.
- **ClassBrowser** — Enables you to browse through C++ source code and libraries, viewing the defined classes and their data and function members. Navigating through your program's classes helps you understand how the classes were defined and how they can be reused.
- **ClassGrapher** — Helps you visualize and understand C++ class hierarchies. It displays a class genealogy tree to identify a program's derived and base classes.

SourceBrowser Base Window

Figure 3-4 shows the SourceBrowser base window.

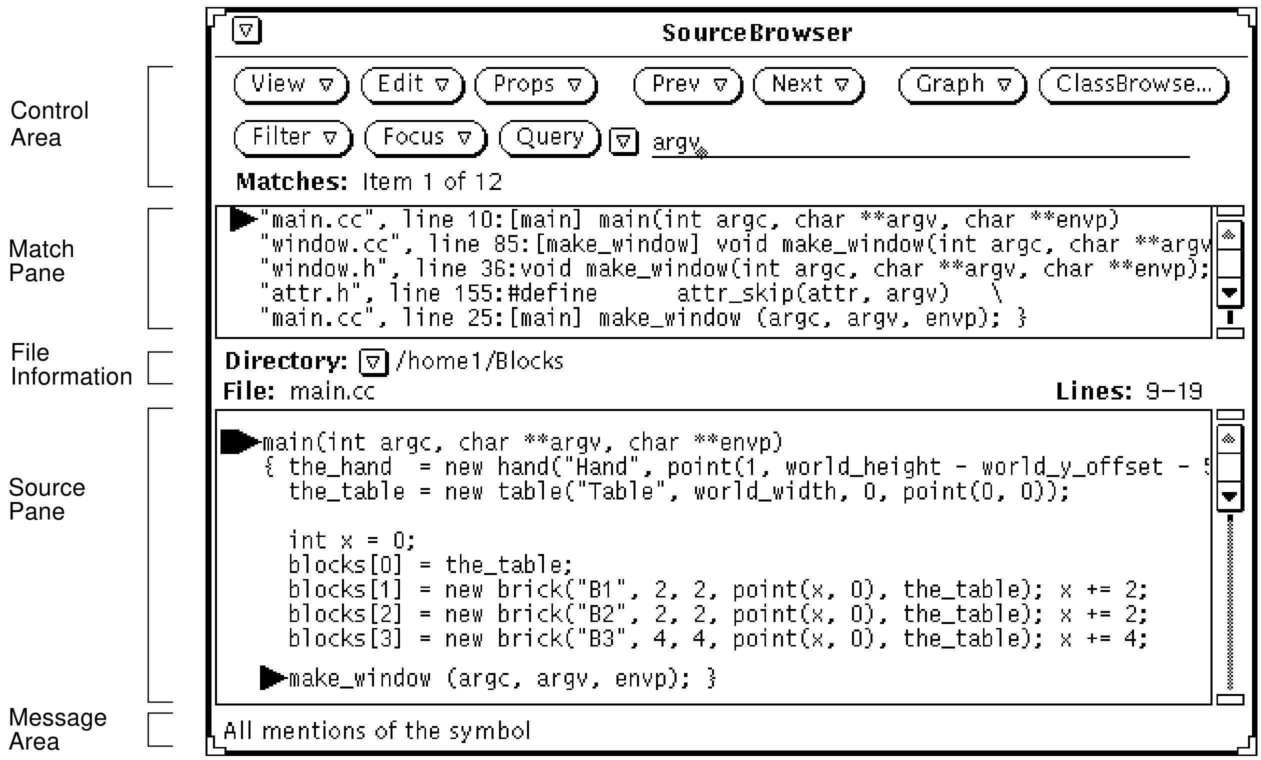


Figure 3-4 SourceBrowser OpenWindows Interface

Besides the standard OpenWindows header, the window contains the following elements:

- A *control area* featuring button menus and a field where you enter text before instructing SourceBrowser to search for matches. The control area also displays a message that reports on the number of matches that result from a query.
- A scrolling *match pane* that displays all matches found by the current query.
- A *file information* area that lists the current working directory, the name of the file being displayed, and the line numbers of the first and last lines being displayed.

- A *source pane* that displays the contents of the source file in which a match was made.
- A *message area* at the bottom of the window that displays the type of search that is the subject of the current query. In Figure 3-4, the type is "All mentions of the symbol," which is shown in the control area to be `argv`.

SourceBrowser Capabilities

This section highlights the features and capabilities of SourceBrowser that are accessible from the base window. Grapher and ClassBrowser capabilities are discussed in *Introduction to SPARCworks*.

- **Finding Symbols and Strings in Source** — In a SourceBrowser session, you issue queries that instruct SourceBrowser to find all occurrences of a specified symbol, string constant, identifier, or search pattern. In response, SourceBrowser displays *matches* (occurrences of the text SourceBrowser was ordered to find) with their surrounding source code.

  SourceBrowser maintains a list of all queries that were conducted during a session. This feature means you can issue a query, view the resulting matches, conduct another query, and then return to the original query with a minimum number of commands.
- **Matching Wildcards and Regular Expressions** — Besides literal text, query patterns can contain *wildcards* (as used in shell-style patterns) or *regular expressions* (as used in `vi`, `grep`, and other SunOS editors and utilities). You specify which convention to use by setting a property in the SourceBrowser Property Window.
- **Performing Semantic Queries** — SourceBrowser can respond to queries that filter matches according to the semantics of the source language. For example, a developer using the C language could issue a very specific query such as "Show all occurrences of the variable *size* when *size* is used as a structure field name." The search can be further focused on specific items or certain classes of code, such as files or functions.
- **Adapting to Different Languages** — SourceBrowser has no built-in, architectural affinity for any specific language. Instead, SourceBrowser obtains information about each supported language by consulting a configuration file. The configuration file expands to incorporate additional languages as new compilers are integrated with SourceBrowser.

- **Editing Source** — You can edit source directly in the text pane. The text pane uses Text Editor commands that are the same as those in other OpenWindows applications such as Mail Tool. Source that is controlled by the Source Code Control System (SCCS) can be checked out quickly from a floating menu in the editing pane.
- **Handling Large Applications** — The first time SourceBrowser runs a query, it creates an index file for use in subsequent queries. The index file enables SourceBrowser to maintain a high level of efficiency regardless of the amount of code it is browsing. SourceBrowser also includes several features specially designed to improve performance while browsing large amounts of code.

SourceBrowser Menus and Commands

Because SourceBrowser presents an OpenWindows interface, you can exercise most of its capabilities using only command buttons and menus. The menus are summarized in Figure 3-5. These menus furnish all the commands you need to:

- View matches, erase matches, edit source code, and set window properties
- Move between matches in the match source panes
- Open the CallGrapher, ClassGrapher, and ClassBrowser windows
- Set a filter and focus
- Issue and view the results of queries

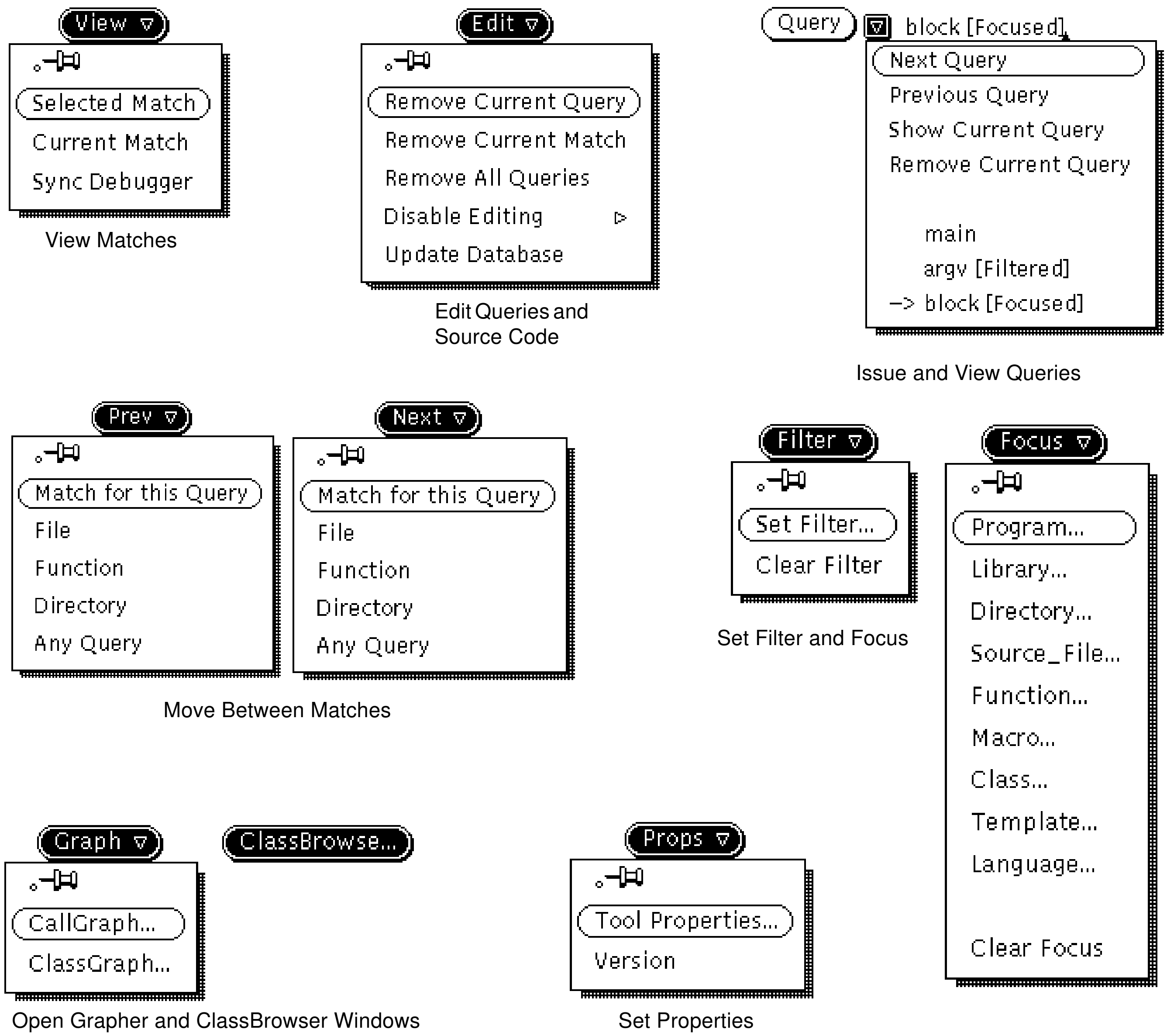


Figure 3-5 SourceBrowser Menus

Besides the menus shown in Figure 3-5, SourceBrowser has a menu (shown in Figure 3-6) that you can display from either the source pane or the match pane. The menu contains often-used items from button menus, and has an additional submenu for performing standard OpenWindows text pane operations.

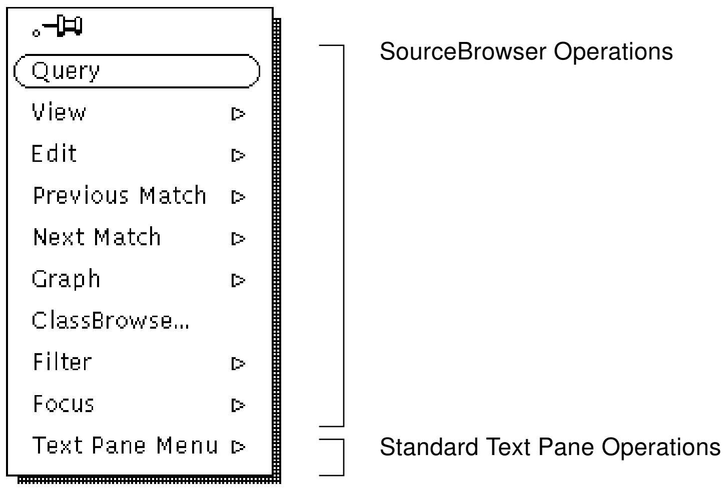


Figure 3-6 SourceBrowser Text Pane Floating Menu

Using SourceBrowser

You initiate a SourceBrowser query in one of two ways:

- By selecting a text string anywhere on the screen and clicking on the Query button
- By typing a string in the text entry field next to the Query button and pressing the Return key

The first method is especially useful when SourceBrowser is used with other OpenWindows programs such as Debugger.

Queries That Return Many Matches

The number of matches the SourceBrowser can handle is limited only by the available swap space. For a symbol like `NULL`, a query might produce many matches. If the number of matches is very large—on the order of several hundred, as shown in Figure 3-7—the information is not very useful.

SourceBrowser attempts to bring some order to the chaos by sorting the matches according to their presumed value to the user: definitions are presented first, then source declarations, and finally external declarations.

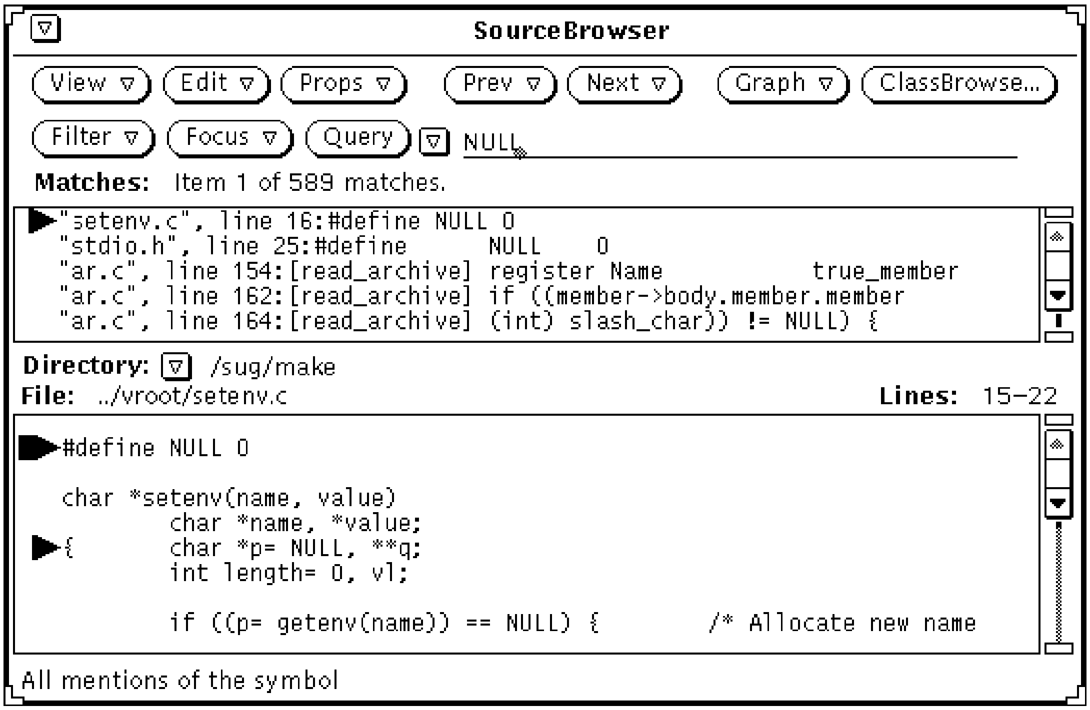


Figure 3-7 Query Resulting in Many Matches

Filtering To Obtain Useful Matches

SourceBrowser filter capabilities can be used to return a more meaningful, smaller number of matches.

You can instruct SourceBrowser to restrict its search on the basis of semantic information specific to a particular language. For example, choosing Set Filter from the Filter button menu opens a pop-up window (shown in Figure 3-8) in which you can specify that the SourceBrowser search only for C-language declarations of the symbol. Subsequent queries will return only matches for declarations. Applying this filter to the example shown in Figure 3-7 results in 14 matches for `NULL` instead of the 589 produced by the All Matches filter.

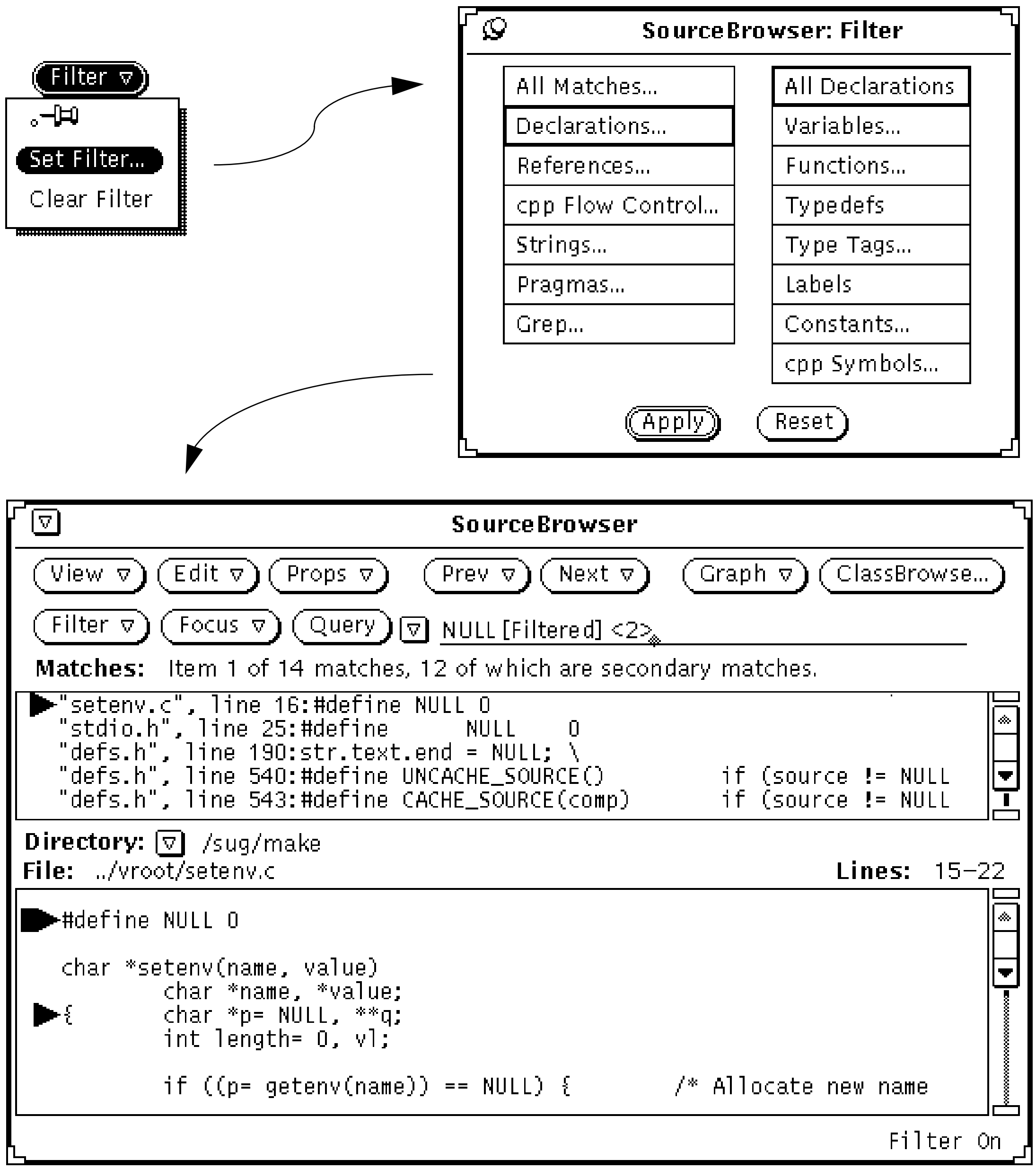


Figure 3-8 Setting Filter for C Language Declarations and Viewing Resulting Matches

Interpreting Secondary Matches

When SourceBrowser matches a string, it provides information about the matches on the Match line. In Figure 3-8, the information reads: "Item 1 of 14 matches, 12 of which are secondary matches."

Secondary matches occur when the compiler is uncertain about how a symbol is used at the time the compiler created the SourceBrowser database. Such uncertainty is common in languages such as C that have macro definitions: the context in which the macro is expanded determines the symbol's usage. A macro is out of context when it is scanned by the compiler. Therefore, the compiler is unable to tell SourceBrowser how the symbols in the macro are used. When SourceBrowser finds such symbols, it reports them as secondary matches and leaves their interpretation to you.

Language-Specific Filtering

The choices on the Filter pop-up window are language-specific and change automatically with the languages that have been compiled into the SourceBrowser database. The choices shown in Figure 3-9 are for C++.

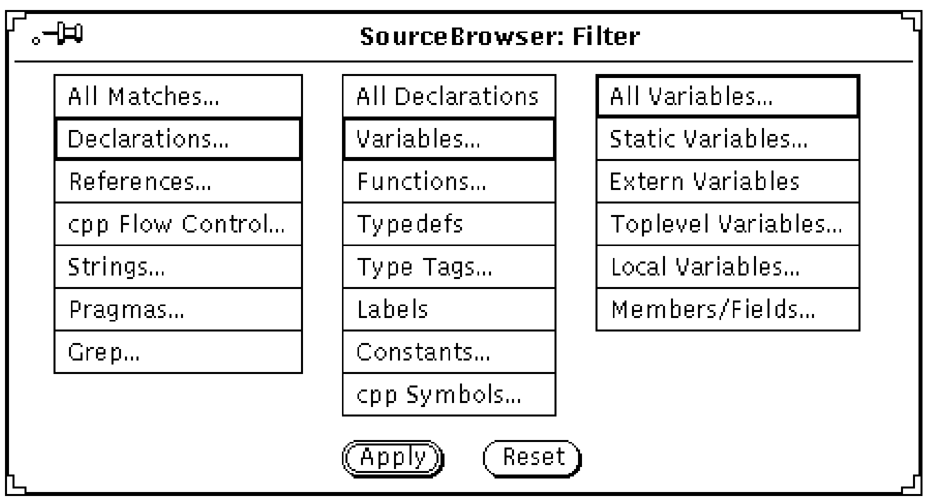


Figure 3-9 Filter Choices for the C++ Language

When a project requires that object files be compiled from sources written in different languages, the linker (`ld`) records this fact in the SourceBrowser database at link time. The Filter pop-up window shows the union of all semantic constructs for the languages that were used to produce the final executable.

Focusing

Besides filtering, you can restrict SourceBrowser to query only certain sections of the source: a specific language, function, or set of source files, for example. This restriction is known as *focusing*.

In Figure 3-10, SourceBrowser has been instructed to focus its queries on the function `hello` and the source file `hello.c`. To set the focus, you select the name of the function and file from lists in Focus pop-up windows and make your choices active. You can activate any number of items in the lists; SourceBrowser thereafter limits its queries to source that is the union of all active items from all Focus lists. With the settings shown in Figure 3-10, a search for the `NULL` symbol would only match text that is either in the function `hello` or the file `hello.c` (or both).

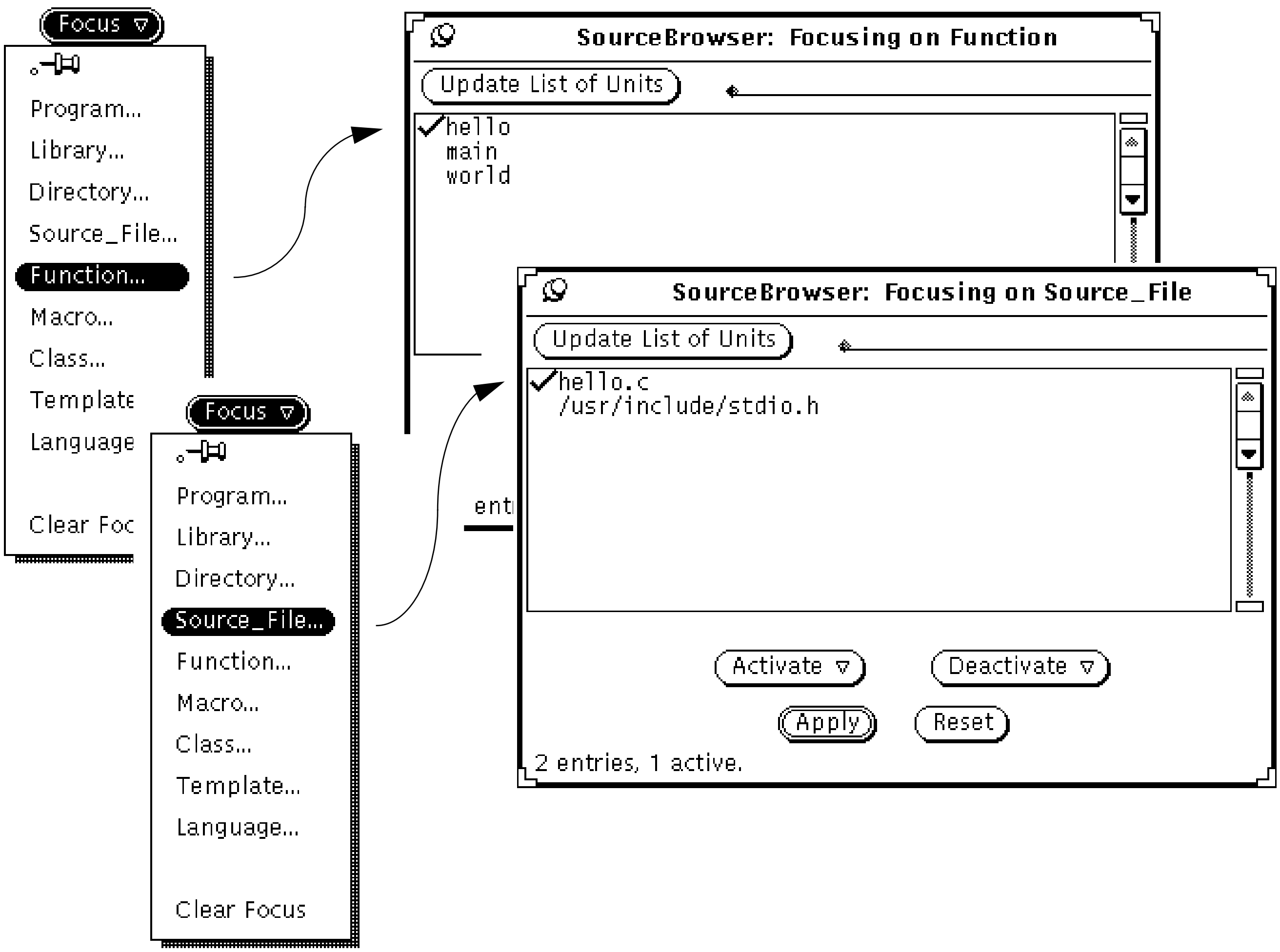


Figure 3-10 Focusing a Query

Matching Strings

The information stored in the SourceBrowser database is determined by the compiler that generates the database (SPARCompilers store strings and symbols). Strings are stored with the quotation marks that are used in the language, which identifies them as strings and enables SourceBrowser to perform queries on them. (To limit a query to strings only, set a filter for Strings.)

Queries for strings are very useful for debugging an application that has printed a message (typically an error message). You can search for a string in the message and find the place in source where the message is printed. In conjunction with Debugger, you can then determine the circumstance that caused the message to be printed and uncover the source of the bug.

The example in Figure 3-11 shows the result of a query for all strings that contain the word `error`. Note that shell-style wildcard characters were used to specify the object of the query: a query for `"*error*"` matches any string in which `error` occurs. Because quotation marks were included in the object of the query, the matches were effectively limited to strings even though no Strings filter was set.

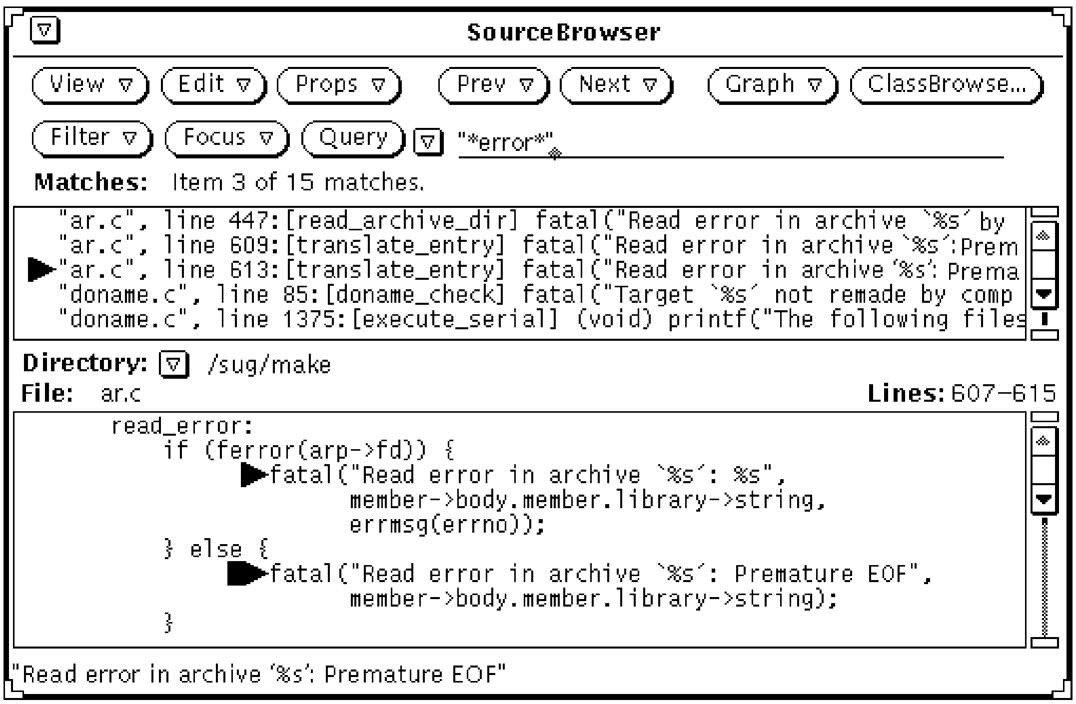


Figure 3-11 Query for an Error Message String

Checking Out Files from SCCS

You can edit source code directly in the SourceBrowser source pane, using OpenWindows Text Editor techniques. A pop-up menu in the source pane enables editing. If the source is controlled by SCCS, the same menu can check out the source file for editing. Edited source files cannot be checked back into SCCS from within SourceBrowser.

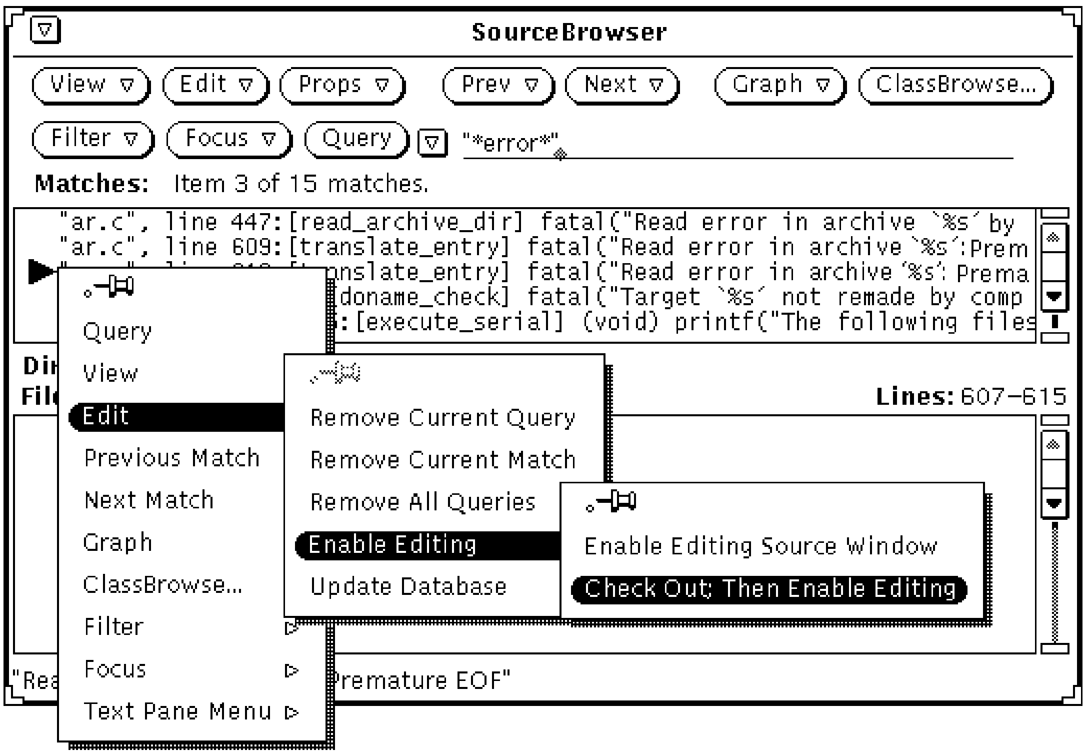


Figure 3-12 Checking Out a File for Editing in the Source Pane

Repeating Queries

As you perform each query, its results are added to a list of all previous queries for any particular session. The abbreviated button next to the Query button displays a menu of past queries that enables you to quickly return to an earlier query without forcing SourceBrowser to reproduce the entire search. When SourceBrowser returns to earlier queries, it also returns to the most recent location in the match list.

Remembering previous queries consumes memory, so SourceBrowser enables you to erase query history when you need to free memory.

Using the Property Window

The SourceBrowser Property Window sets characteristics of the SourceBrowser display. Such features as the ability to turn off case sensitivity for searches (useful for FORTRAN programmers) are found here. By setting the "Show Match With" property to "File Name and Line Number Only," you can improve performance by reducing the amount of information displayed for each match.

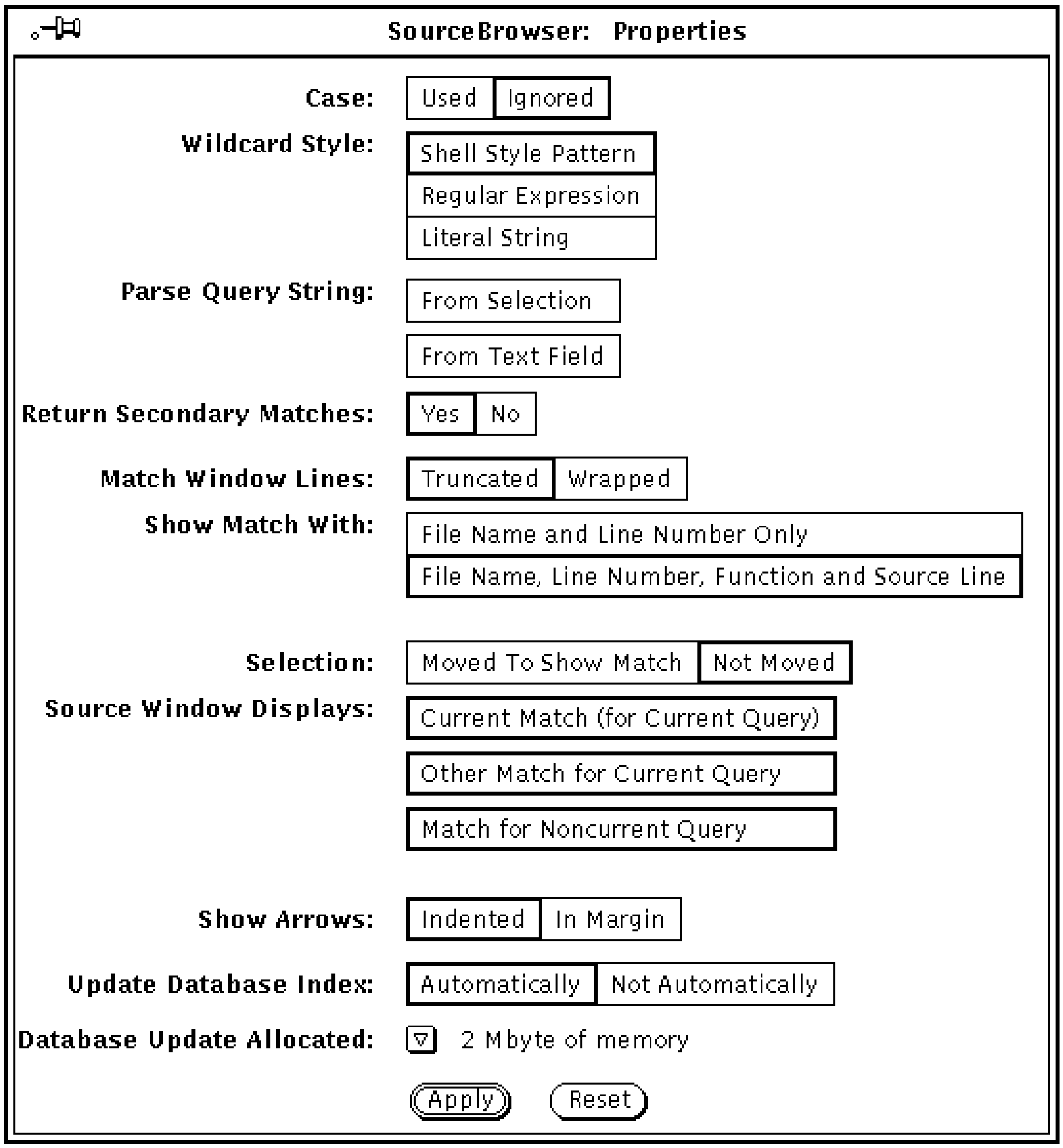


Figure 3-13 SourceBrowser Property Window

SourceBrowser Data Flow Overview

Figure 3-14 shows how the compiler tools create the SourceBrowser database. As the compiler digests source files, it translates them into object (`.o`) files and creates SourceBrowser database (`.bd`) files. The assembler and linker also produce database files that describe their actions. The compiler tools must exit without errors or warnings to ensure error-free `.bd` files. However, even a compile that completes with errors can yield `.bd` files that contain much valid, useful information, and these files can be used by SourceBrowser.

When SourceBrowser responds to the first query in a session, it creates an index file to speed its matching activities. SourceBrowser thereafter consults the index and the `.bd` files to perform matches. In addition, it reads a configuration file (with a `.ex` suffix) that provides information about the language of the source it is browsing.

SourceBrowser database files are independent of architecture. Thus, files created on a Sun-3 machine can be browsed by SourceBrowser running on a SPARCstation machine.

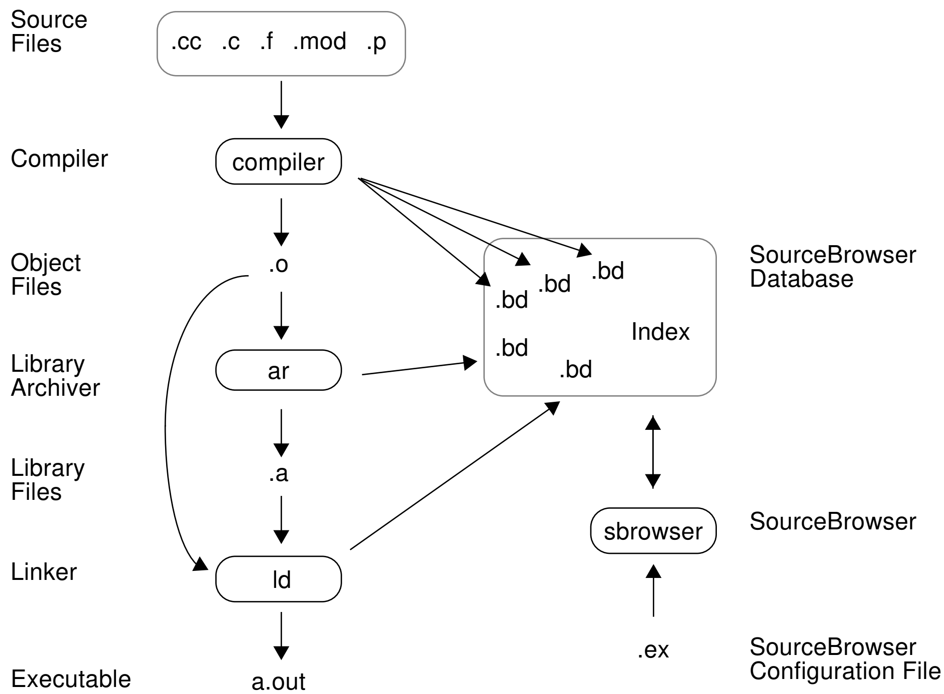


Figure 3-14 Overview of SourceBrowser Data Flow

Structure of SourceBrowser Database

Unless you specify otherwise, the SourceBrowser database is created in the directory in which you execute the compile command (the current working directory). This scheme ensures that the compiler (and SourceBrowser, when it creates its index file) has permission to write in the directories where the database files are created. SourceBrowser can browse the database from another directory, but you must place a symbolic link to the database in the directory where SourceBrowser is invoked.

The database structure is built beneath a single directory named `.sb`. This directory contains subdirectories that hold the `.bd` files and the single index file. The compiler creates one `.bd` file per source file. When you browse with SourceBrowser you are actually browsing the database files, which completely describe the source files. If you enable editing and change a source file, SourceBrowser keeps track of the changes in its database to reflect the new location of matches.

Handling Large Projects

SourceBrowser can browse extremely large projects for which source files are not contained in a single directory. You can take one of two approaches:

- Keep the database in a single directory that is referenced by symbolic links from source subdirectories where SourceBrowser is used.
- Partition the database and inform SourceBrowser of the location of database files through an initialization file.

SourceBrowser Resource Requirements

SourceBrowser requires system resources during compilation, for database storage, and while querying the database. This section describes the resources required.

- **Disk Space** — The database files take up approximately the same disk space as the source files from which they are created.
- **Compiler Performance Penalty** — Creating the SourceBrowser database exacts a performance toll from the compiler. For example, a build that uses `cc` slows down an average of 15% while creating the database.
- **Query Time—Building the Index** — The first time you run a query, SourceBrowser builds an index file. During this process, SourceBrowser must open all the `.bd` files, so the time required to build the index file is directly proportional to the number of `.bd` files present. The actual time required to build the index file depends on the input/output system at your site. If you run over the network, for example, the process is more time-consuming than if you run on a fast local disk.

  Developers who use `make` to build programs can create the index file during the build process by placing a special target in their makefiles.
- **Query Time—After Building the Index File** — The SourceBrowser takes advantage of the index file so that the time required for a query process is independent of the size of the database. Query time is, however, directly

proportional to the number of source files (actually, the number of `.bd` files) on which SourceBrowser is focused, and directly proportional to the number of matches SourceBrowser finds.

Typical queries (those that return fewer than 30 matches) run in two or three seconds on a SPARCstation 1+.

- **Limits to Performance** — There are no limits in the SourceBrowser architecture on the number of matches it can return. However, queries that return large numbers of matches (hundreds or thousands) require large amounts of disk swap space.

CallGrapher

The SourceBrowser base window provides a textual view of source. For occasions when a graphical view is more valuable, selecting an item from the Graph menu brings up one of two graphical displays: CallGrapher for function call graphs and ClassGrapher for C++ class graphs.

Nodes in the CallGrapher graph represent functions; directed lines between the nodes (called *edges* or *arcs*) represent calls.

Because `main()` can be thought of as the root function in a C or C++ program, an expedient way to access the function call tree is to query for `main()` in the SourceBrowser base window. Select `main()` when it is found, and then bring up its function call graph by choosing CallGraph from the Graph menu. The resulting function call graph will show all of the functions called by `main()`. Figure 3-15 shows the result of this process.

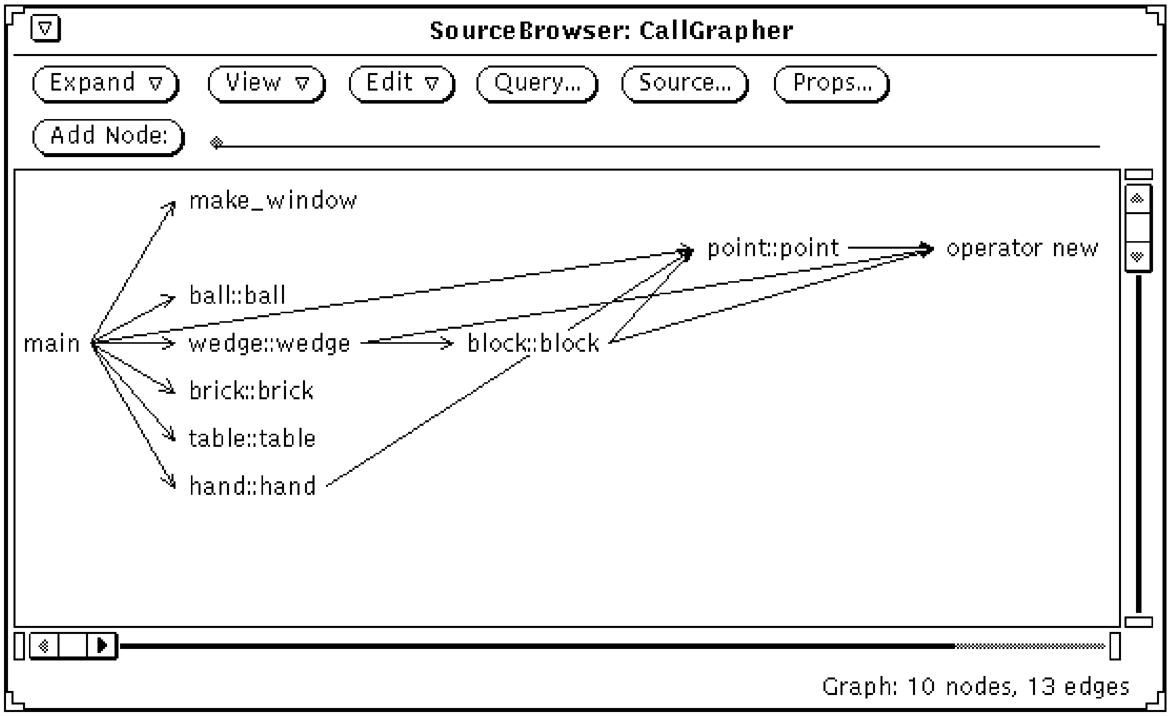


Figure 3-15 Function Call Tree Rooted at `main()`

CallGrapher has too many functions and features to discuss in detail in this overview — the main features are summarized below (for details, see the manual *Browsing Source Code*, part number 800-6570-11):

- You can expand and contract the graph by displaying additional functions or restricting the graph to a subset of the overall graph. For example, by double-clicking on one of the functions in the graph you can show all the functions it calls. In fact, this is a convenient method for navigating through the graph.
- You can add, find, and delete functions in the graph.
- You can issue a query on a function in the call graph and view a function with its surrounding source code in the SourceBrowser or Debugger window. This feature makes moving from a graphical to a textual view and back again very easy.

As an aside, note that CallGrapher can locate hidden function calls in C++ code that are not apparent to SourceBrowser. SourceBrowser does not reveal hidden function calls because it is built on a "what-you-see-is-what-you-browse" model — and, since hidden function calls are not stated explicitly, they cannot be found in source code. However, CallGrapher uses information in the SourceBrowser data base that gives it access to these calls, under the condition that operators in a user-defined class are also user-defined.

To illustrate this conditional constraint, consider the following C++ code fragment:

```
#include <iostream.h>
class Integer{
 int value;
public:
 Integer(int i) : value(i) {};
 Integer() : value(0) {};
 operator int() { return value; };
};

main() {

Integer x;

x = 10;    //  CallGrapher finds
           //  Integer::Integer(int) hidden here,
           //  but not
           //  Integer::operator =(Integer &)

cout << x; //  CallGrapher finds
           //  Integer::operator int() hidden here.
 }
```

Note that the CallGrapher cannot find the call

```
Integer::operator =(Integer &)
```

hidden in the statement

```
x = 10;
```

because the class `Integer` that has been defined in the code fragment does not include its own definition of the operator =, but instead uses the default allowed by C++ for this instance. When the code is modified to include a user definition of operator =, the hidden call is shown in CallGrapher.

ClassGrapher

ClassGrapher is like CallGrapher, but it shows a genealogy tree for learning about classes in the C++ language.

In the same way you would select the `main()` function as the root of the CallGrapher function call tree, in ClassGrapher you would select the base (root) class. In the ClassGrapher window shown in Figure 3-16, `ios` was selected as the root class, and the graph shows how other classes are derived from it. (Note the multiple inheritance characteristics of `fstream`, which is derived from both `fstreambase` and `iostream`.)

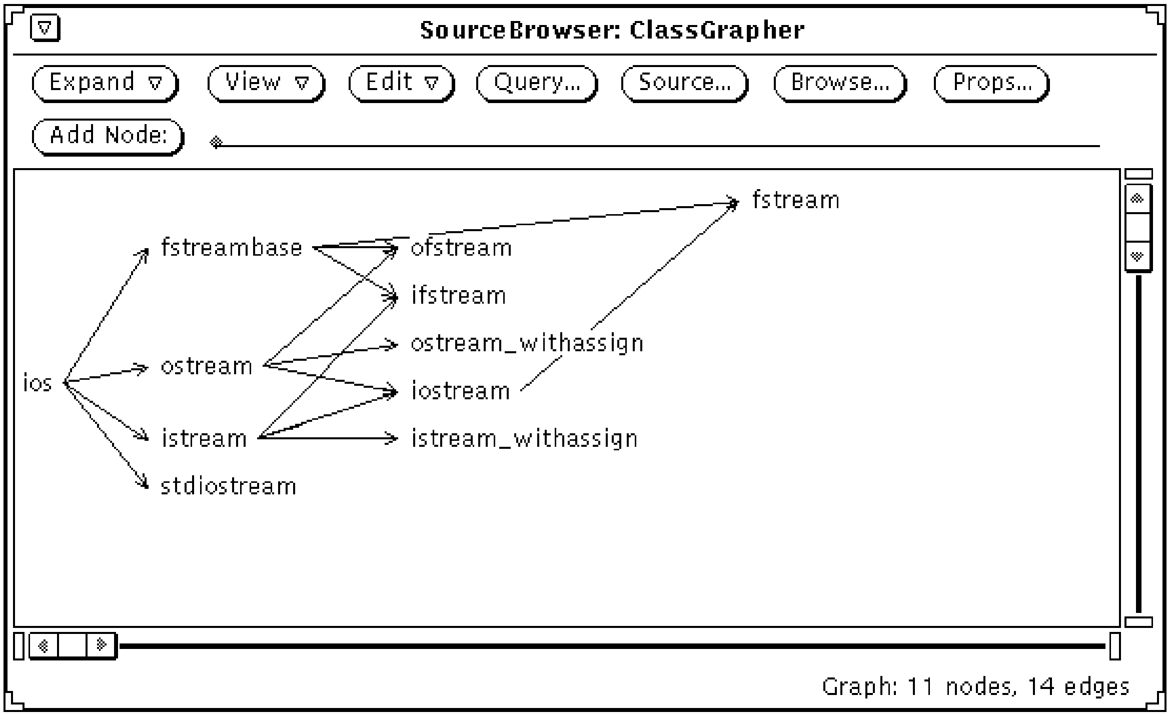


Figure 3-16 ClassGrapher Window

Showing an entire C++ class library hierarchy at one time is a two-step process: First, set the expansion depth in the Properties window to a number at least as large as the maximum number of generations of classes in your program (the default value for this number is 1). Then, double-click on the base class of the hierarchy.

The ClassGrapher has most of the same graphing features as CallGrapher, including the ability to issue a query on a class in the graph and then view the class with its surrounding source code in the SourceBrowser source pane. You can also activate the ClassBrowser from the ClassGrapher.

ClassBrowser

You can activate ClassBrowser from either the SourceBrowser or ClassGrapher window. Either way, you select the name of a class and bring up ClassBrowser, which then shows the data members and member functions defined in the class you select — both those defined on the selected class and (optionally) those inherited from ancestor classes.

In Figure 3-17, the node `istream` was selected in ClassGrapher, then ClassBrowser was started. ClassBrowser opened with `istream` loaded as the current class, with its data and function members identified. By default, ClassBrowser displays all private, protected, and public members of a selected class. You can also list the inherited members and friend functions of the selected class.

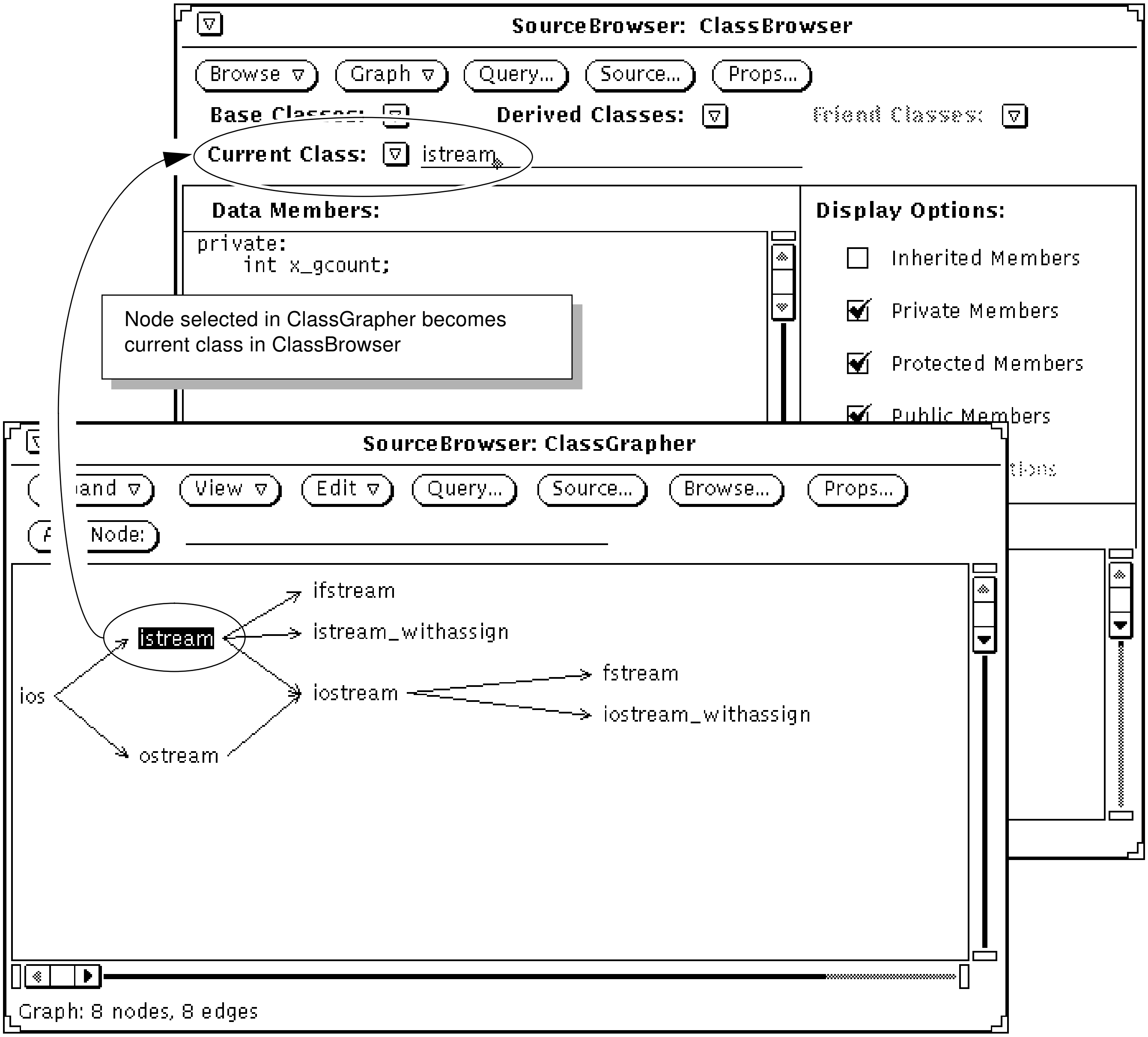


Figure 3-17 Starting ClassBrowser from CallGrapher

Virtual functions are identified by glyphs in ClassBrowser. This feature lets you know which functions will be chosen at run time and which will be chosen at compile time. A different glyph identifies in-line functions.

From ClassBrowser you can view source for a class or a member function by selecting its name and clicking on the Source button, either in SourceBrowser or Debugger (as controlled by a ClassBrowser property setting). From ClassBrowser you can also select a symbol and run a query on it (in SourceBrowser) to see all the places in source where the symbol appears.

To browse several classes at once, start a separate instance of ClassBrowser for each class you want to browse (by selecting the In New Window item from the Browse button menu on the ClassBrowser base window).

ClassBrowser and ClassGrapher contain many features to assist you in navigating through complex class structures. For example, ClassBrowser maintains a dynamic menu of previously browsed classes for easy return to those classes. Also, you can automatically graph a class when it becomes the currently browsed class.

How SourceBrowser, the Graphers, and ClassBrowser Interrelate

Figure 3-18, adapted from *Browsing Source Code*, shows how the major SourceBrowser windows interrelate. The ClassBrowser and Grapher windows are activated from the SourceBrowser base window. You can also activate ClassBrowser from ClassGrapher and vice versa. The figure does not show Debugger, which, along with SourceBrowser, can display the source code of a class or function selected in a Grapher or in the ClassBrowser.

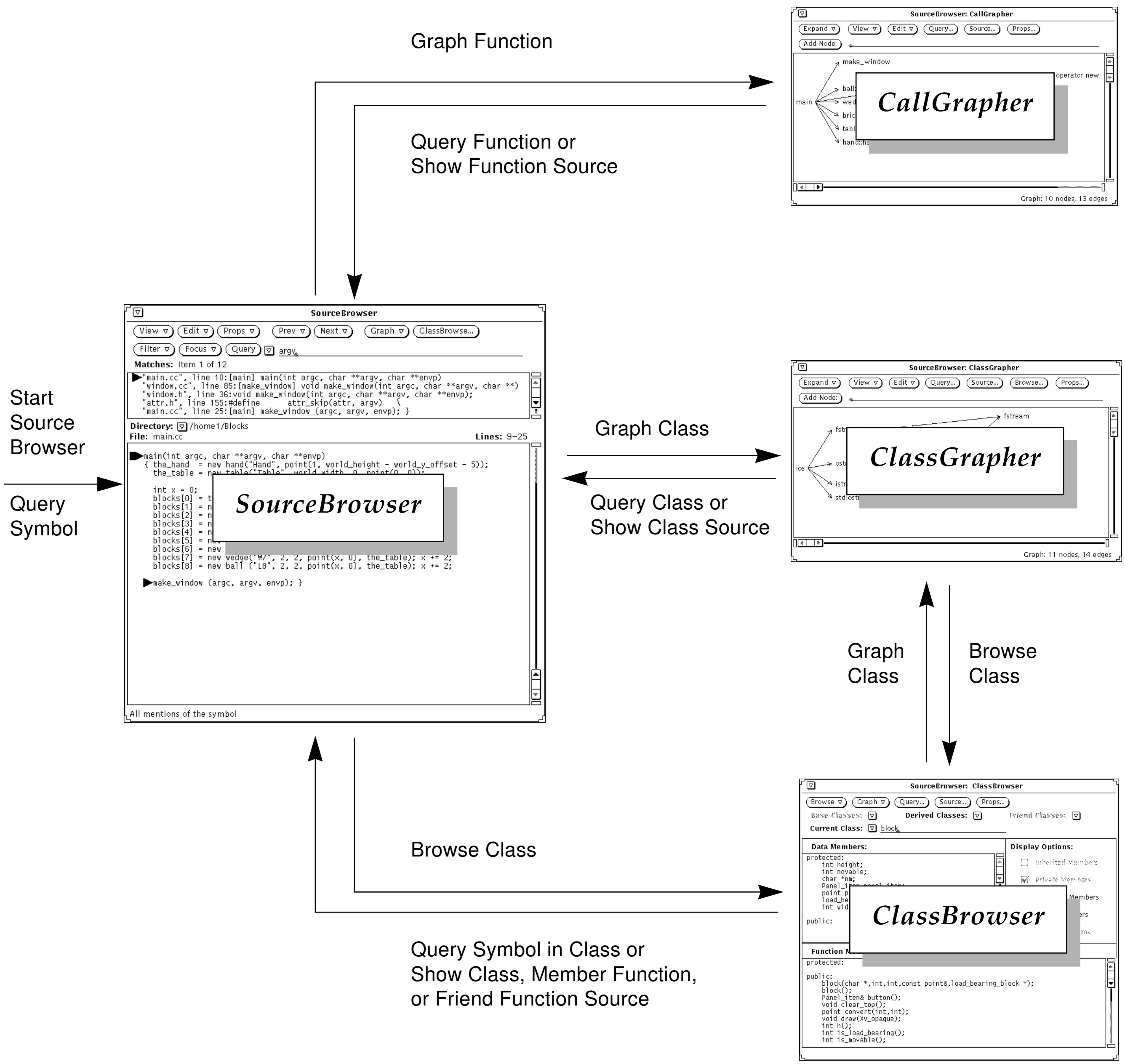


Figure 3-18 How SourceBrowser, the Graphers, and ClassBrowser Interrelate

Software Release Versions 4

Current Sun Software Products and Release Versions

The following tables contain lists of Sun software products and their respective release versions, current as of November 30, 1992. These tables appear monthly in the STB for your convenience.

Please contact your sales representative if you would like to update the release level of a Sun software product you already use, or wish to purchase another product. Use the tables to determine whether your release is the most current.

Please note the following software product change(s):

a. As of the release of Solaris 2.0, the C compiler will be unbundled.

Operating Systems

Product Name	Current Release
SunOS (Sun-4 architectures)	5.0
SunOS (Sun-3 architectures)	4.1.1_U1

Window Systems & Desktop Utilities

Product Name	Current Release
OpenWindows	3.0
SimplifySQL	1.1

Graphics Software

Product Name	Current Release
SunVision	1.2
XGL Graphics Library	2.0
SunGKS	4.1
SunPHIGS	2.0
SunPEX	1.0
Direct Xlib	1.0

Software Development

Product Name	Current Release
SPARCworks	2.0
SPARCompiler C *	2.0
SPARCompiler C++ *	3.0
SPARCompiler FORTRAN *	2.0.1
SPARCompiler Pascal *	3.0
Sun Ada Development Environment	1.1
Sun COBOL	1.0
Sun COBOL Runtime System	1.0
Sun COBOL Forms-2	1.0
NetISAM	1.0
OpenWindows Developer's Guide (DevGuide)	3.0.1
ONC / RPC Application Toolkit	1.0

* Available in both SPARCworks Professional and SPARCompiler versions

Software product availability is current for U.S. releases.
Availability may be delayed outside the U.S.

System Management

Product Name	Current Release
Online: Backup Copilot	1.0
Online: DiskSuite	1.0
Online: IPI Dual Port Tools	1.0
Sun SHIELD ARM and ASET	1.0
SunNet Manager	2.0
Sun Database Excelerator (SunDBE)	1.3

Connectivity Products

Product Name	Current Release
SunLink CG3270	7.1
SunLink SNA3270	7.0.1
SunLink BSC3270	6.1
SunLink BSCRJE	6.0
SunLink SNA Peer-to-Peer	7.0.1
SunLink Local 3270	6.1.1
SunLink Channel Gateway	7.1
SunLink DNI	7.0
SunLink TE320	7.0
SunLink OSI	7.0.1
SunNet MHS	7.0
SunLink IR	6.0
SunLink X.25	7.0
SunLink TRI/S	2.0
SunLink HSI/S	1.1
SunLink FDDI/S	1.0
Serial Parallel Controller	1.1
SunLink MCP	6.0
SunLink FDDI/DX	2.0
SunLink NetWare	1.0

Software product availability is current for U.S. releases.
Availability may be delayed outside the U.S.

Other Products

Product Name	Current Release
NeWSprint	2.0
VideoPix	1.0
PC-NFS	4.0
PC-NFS LifeLine	2.0
PC-NFS Advanced Telnet	1.0
PC-NFS Programmer's Toolkit	4.0
SunPC	3.0

Software product availability is current for U.S. releases.
Availability may be delayed outside the U.S.

Updated Sun Operating System Support Matrix

When a platform has been designated as a final release, Sun's policy is to provide hardware and software support for out-of-production or transitioned systems and the latest compatible software for those systems for a minimum period of five years from the last order date.

As an ongoing policy, the level of support for SPARC products will generally decline at an increasing rate as new operating system releases are made available. Typically, the two most current versions of the operating system will be supported at level 1.

The details of Sun's software support policy have been published in a Sun Operating System Support matrix in past STB issues. As illustrated in the support matrix, older versions of the operating system generally receive lower levels of software support than newer versions. Thus, we recommend that you upgrade your system to the most current versions to ensure you receive the best quality of support from Sun.

Since we continually enhance the operating system to meet our customers' needs, we promised to keep you abreast of updates to the support matrix via the STB. Now that SunOS 4.1.3 and Solaris 2.0 are available, we provide an updated matrix.

Sun uses "support levels" to describe its software support policy. Please note that the services described in these levels are subject to local availability—contact your local service manager should you have any questions.

Level 1 Support

Level 1 offers the complete array of services contained in Sun's Personal AnswerLine Service, AnswerLine Service, and Software Update Service (see Note ①). Level 1 support also includes the full availability of options such as SunStart™, Sun's educational services, and supplementary consulting products and services.

Level 2 Support

Level 2 includes Sun's Personal AnswerLine and AnswerLine levels of phone support, limited availability of software and documentation, bug report submission and tracking, and emergency fixes and workarounds (see Note ①). Level 2 support also includes availability of supplementary consulting products and services.

Level 3 Support

Level 3 provides Time and Materials (T&M) support only.

Note ① — Software updates will only be available for the most current release. When a version of software has been designated as the final release for a platform, only critical bug fix releases will be available for that platform.

Sun Operating System Software Support Matrix

The following tables illustrate Sun's OS software support matrix. See Note ② below regarding this section.

For Sun-2	3.5	4.0/4.0.1	4.0.3	(see Note ③)
1/91-1/92	level 3	level 3	level 2	
1/92-1/94	level 3	level 3	level 3	

For Sun-3	3.5	4.0/4.0.1	4.0.3/4.0.3e	4.1	4.1.1	(see Note ③)
1/91-1/92	level 3	level 3	level 2	level 1	level 1	
1/92-1/95	level 3	level 3	level 3	level 2	level 2	
1/95-1/96	level 3	level 3	level 3	level 3	level 3	

For Sun386i	4.0.1i	4.0.2i	(see Note ③)
1/91-1/92	level 2	level 1	
1/92-1/95	level 3	level 2	
1/95-1/96	level 3	level 3	

For SPARC	3.5	4.0/4.0.1	4.0.3/4.0.3e	4.1	4.1.1	4.1.2	Solaris 2.0	4.1.3
1/91-present	level 3	level 3	level 3	level 2	level 2	level 1	level 1	level 1

Note ② — The above levels 1 and 2 are for contract customers only. Non-contract customers have the option to purchase support on a Time and Materials basis or use Sun's consulting services. The then-current unbundled products will be supported at the same level as the customer's operating system. Support for emergency fixes and workarounds may require the customer to upgrade to a more current version of the product.

Note ③ — These will be the final releases for these product lines, with the exception of critical bug fix releases. Sun's future SVR4-based operating system will function primarily on SPARC processor-based systems (Intel-based support for SVR4 is available with SunSoft's INTERACTIVE UNIX product line). Interoperability between Sun-3 or Sun386i systems and SPARC SVR4 systems will be fully supported through services such as NFS, NIS, remote login, and the like.

In addition, Sun is committed to supporting Sun-3 diskless clients running SunOS 4.1.1 on Solaris 2.0 servers.

Notes & Comments 5

World Hotlines

Sun Customers throughout the world have service hotlines available for both software and hardware support questions. The service hotlines are shown below. If your country is not shown in the table, please phone your local Sun sales office. World hotlines are divided into those for the U.S.A., Europe, and Intercon.

United States

All	**1-800-USA-4SUN** (**1-800-872-4786**)

Europe

Finland	Sun Microsystems Oy	**+358-0-502-2700**
France	Sun Microsystems France SA	**+33-1-30-67-50-80**
Germany	Sun Microsystems GmbH	**+49-89-46008-321**
Italy	Sun Microsystems Italia SpA	**+39-6055228**
Netherlands	Sun Microsystems Nederland BV	**+33-501234**
Spain	Sun Microsystems Iberica SA	**+34-1-5560243**
Sweden	Sun Microsystems AB	**+46-8-623-9100**
Switzerland	Sun Microsystems (Schweiz) AG	**+41-1-825-7272**
	Sun Microsystems (Suisse) SA	**+41-22-364-3639**
United Kingdom		
	Sun Microsystems UK Ltd	**+44-276-691974**

Intercon

***Canada*:** Quebec District

***Canada*:**	Quebec District Montreal, Quebec	**(514) 744-9240**, **(800) 361-1554**
	Ottawa District Ottawa, Eastern Canada	**(613) 723-8112**, **(800) 465-4415**
	Central District Toronto, Central Ontario	**(416) 490-9490**, **(800) 263-1680**
	Prairies District Manitoba, Saskatchewan, Alberta, Calgary, Western Canada	**(403) 262-6722**, **(800) 661-9256**
	Edmonton District Edmonton	**(403) 421-7264**, **(800) 661-3643**
	B. C. District Vancouver, British Columbia	**(604) 684-4120**, **(800) 663-0440**
	National CSS/Education	**(416) 477-6745**

Australia	Sun Australia	**(008) 02-4417**
New Zealand	Auckland Wellington	**64-9-525-2749** **64-4-499-2344**
Hong Kong	Sun Asia	**852-802-4188**
Japan	Nihon Sun	**044-819-3121**

Latin America: Contact the Latin American hotline at ICON headquarters:
U.S. Office **(415) 688-9160**

Within ***Brazil***, you may also contact:
Sun do Brasil **55-11-887-9011**

Caribbean: Call your local distributor.

All Other Countries
Call **(415) 688-9372**, which is a U.S. number.

Reporting Bugs

This article contains three sections for submitting bugs. The first section describes procedures to use within the United States and Canada. The second section describes procedures within Europe. The third section describes procedures within ICON.

Submitting Software Bugs: United States and Canada

This section contains information on reporting bugs within the U.S., for customers holding and not holding support contracts.

Sun's United States Solution Center accepts software bug reports from Sun users via electronic mail and by phone. The method you use to submit a bug report varies with your needs.

U.S. users holding support contracts can report bugs to US Solution Center via the **(800) USA-4-SUN** phone hotline. Canadian users holding support contracts should call their local support center. The U.S. Solution Center phone hotline is the fastest way for a customer to find out if a problem is known and if a workaround exists. The status of previously-reported bugs can also be obtained in this way.

U.S. customers holding support contracts can also submit bug reports electronically to the address *sun!hotline* (or *hotline@Sun.COM*). Canadian customers need to use *hotline@Canada.Sun.COM*. This method generates a service order, and can be used when lines of code or other information difficult to relay over the phone is needed to describe the bug.

Customers who do not hold Sun software support contracts can report bugs via electronic mail to the address *sun!sunbugs* (or *sunbugs@Sun.COM*). These reports are reviewed periodically to determine proper disposition. Those reports determined to be from supported customers are forwarded to the U.S. Solution Center for handling. Reports from customers who cannot be verified as holding a support contract are reviewed by Sun's engineering and support personnel. An internal bug report is generated if the reported bug is new and verifiable.

Finally, customers not holding software support contracts may call the **(800) USA-4-SUN** phone hotline to report a problem and request support on a Time and Materials (T&M) basis. Canadian customers should call their local support center. In this case, please have a Purchase Order (PO) number for billing purposes.

Summary: United States and Canada

For U.S. contract customers, **(800) USA-4-SUN** is the best method to report bugs. Canadian contract customers should report bugs to their local support center. The hotlines to use appear monthly in the note entitled "World Hotlines."

The electronic mail address *sun!hotline* (or *hotline@Sun.COM*) is available for U.S. customers to submit materials that are difficult to relay over the phone. Canadian customers should use *hotline@Canada.Sun.COM* .

For non-contract customers, the electronic mail address *sun!sunbugs* (or *sunbugs@Sun.COM*) is available to report bugs. To help us serve you better, please include the following information with all electronic mail reports:

- Your name
- The name and address of your organization
- Your Sun site code, if available
- Your workstation model and serial number
- The software release(s) you are running
- A description of the problem that you are experiencing

Submitting Software Bugs: Europe

This section contains information on reporting bugs within Europe, for customers holding and not holding support contracts.

Procedures for submitting bugs are similar to those used in the United States. All customers should use their local country Answer Center to report bugs, with contract customers receiving a specific follow-up.

Sun customers not holding software service contracts can call their local Answer Center, and will need to provide a Purchase Order (PO) number at the time of the call.

Summary: Europe

To help European service centers serve you better, please include the following information with all electronic mail reports:

- Your name
- The name and address of your organization
- Your Sun site code, if available
- Your workstation model and serial number
- The software release(s) you are running
- A description of the problem that you are experiencing

Finland

Report bugs to the Finland Answer Center at the postal address shown below.

Hotline	Hotline Telephone: +358-0-502-2700
Sun Microsystems Oy	Telefax: +358-0-502-2800
Sinikalliontie 12	
SF-02630 Espoo, Finland	E-mail Address: *Hotline@Finland.Sun.COM*

Special Dispatch Arrangements: Provide Dispatch with the Contract number.
Arrangements for Non-Contract Customers: Please provide a valid PO number for billing on a Time and Materials (T&M) basis.

France

Report bugs to the France Answer Center at the postal address shown below.

Service Hot Line	Hotline Telephone: +33-1-30-67-50-80
Sun Microsystems France SA	Telefax: +33-1-30-67-53-70
13, Avenue Morane Saulnier	Minitel: 36 14 code SUNFRANCE
BP 62	
78142 Velizy, France	

Special Dispatch Arrangements: Please provide Dispatch with System serial number *or* Contract number.
Arrangements for Non-Contract Customers: Please provide a valid PO number for billing on a Time and Materials (T&M) basis, and order this support at the above address.

Germany

Report bugs to the Germany Answer Center at the postal address shown below.

Hotline	Hotline Telephone: +49-89-46008-321
Sun Microsystems	Telex: 5218197 sun
Bretonischer Ring 3	Telefax: +49-89-46008-320
8011 Grasbrunn, Germany	E-mail Address:
	{sunuk,unido}!sunde!hotline or
	hotline@Germany.Sun.COM

Arrangements for Non-Contract Customers: Please provide a valid PO number for billing on a Time and Materials (T&M) basis.

Italy

Report bugs to the Italy Answer Center at the postal address shown below.

Hotline	Telephone: +039-6055228
Sun Microsystems Italia SpA	Telex: 326423 MBOX SUNITA
Centro Direzionale Colleoni	Telefax: +039-6056764
Palazzo Andromeda (Ingresso 1)	
Via Paracelso 16	
20041 Agrate Brianza	
Milan, Italy	

Arrangements for Non-Contract Customers: Please provide a valid PO number for billing on a Time and Materials (T&M) basis.

The Netherlands

Report bugs to The Netherlands Answer Center at the postal address shown below.

Sun Microsystems Nederland BV	Hotline Telephone: +33-501234
Computerweg 1	Telefax: +33-553058
3821 AA Amersfoort	Telex: 79297 sun nl
Netherlands	

Arrangements for Non-Contract Customers: Please provide a valid PO number for billing on a Time and Materials (T&M) basis.

Spain

Report bugs to Spain Answer Center at the postal address shown below.

Sun Microsystems Iberica SA	Hotline Telephone: +34-1-5560243
Plaza Pablo Ruiz Picasso s/n	Telefax: +34-1-5564097
Torre Picasso Planta 25	
28020 Madrid, Spain	

Arrangements for Non-Contract Customers: Please provide a valid PO number for billing on a Time and Materials (T&M) basis.

Sweden

Report bugs to the Sweden Answer Center at the postal address shown below.

Sun Microsystems AB
Box 51
S-164 94
Kista, Sweden

Hotline Telephone: +46-8-623-9100

E-mail Address: *hotline@Sun.se* or *hotline@Sweden.Sun.COM*

Arrangements for Non-Contract Customers:
Please provide a valid PO number for billing on a Time and Materials (T&M) basis.

Switzerland

Report bugs to the Switzerland Answer Center at one of the postal addresses shown below.

(*German*)
Sun Microsystems (Schweiz) AG
Postfach
Eschenstrasse 8
CH-8603 Schwerzenbach, Switzerland

Telephone: +41-1-825-7272
Telefax: +41-1-825-7222
E-mail: *hotline@Swiss.Sun.COM*

(*French*)
Sun Microsystems (Suisse) SA
Route des Avouillons 12
CH-1196 Gland, Switzerland

Telephone: +41-22-364-3639
Telefax: +41-22-364-1176
E-mail: *answerline@Swiss.Sun.COM*

Special Dispatch Arrangements: Provide Dispatch with the Contract number and System serial number.
Arrangements for Non-Contract Customers: Please provide a valid PO number for billing on a Time and Materials (T&M) basis.

United Kingdom

Report bugs to the UK Answer Center at the postal address shown below.

Sun Microsystems UK Ltd
Watchmoor Park
Riverside Way
Camberley, Surrey
GU15 3YL, England

Hotline Telephone: 0276-691974
Telefax: 0276-681546

Special Dispatch Arrangements: Please provide Dispatch with either the System serial number *or* Contract number.
Arrangements for Non-Contract Customers: Please provide a valid PO number for billing on a Time and Materials (T&M) basis.

Submitting Software Bugs: Intercon

This section contains information on reporting bugs within Intercon, for customers holding and not holding support contracts.

Procedures for submitting bugs are similar to those used in the United States. All customers should use their local country Answer Center to report bugs, with contract customers receiving a specific follow-up.

Sun customers not holding software service contracts can call their local Answer Center, and will need to provide a Purchase Order (PO) number at the time of the call.

Summary: Intercon

To help Intercon service centers serve you better, please include the following information with all electronic mail reports:

- Your name
- The name and address of your organization
- Your Sun site code, if available
- Your workstation model and serial number
- The software release(s) you are running
- A description of the problem that you are experiencing

Detailed information for individual countries follows.

Australia

Report bugs to the Australian Answer Center at the postal address shown below.

Hotline	Hotline Telephone: (008) 02-4417
Sun Microsystems Australia Pty. Ltd.	Telefax: 02 436 1084
PO Box 320	E-mail Address: *hotline@Aus.Sun.COM*
Artarmon, New South Wales 2064	
Australia	

Special Dispatch Arrangements: Please provide Dispatch with either the System serial number *or* Contract number.

Arrangements for Non-Contract Customers: Please provide a valid PO number for billing on a Time and Materials (T&M) basis.

Japan

Report bugs to the Nihon Sun Answer Center at the postal address shown below.

Nihon Sun Microsystems K.K
KSP Technical Center - Answer Center
KSP R&D Bldg. B-1F-112
Sakato 100-1, Takatsu-ku,
Kawasaki, Kanagawa, 213
Japan

Hotline Telephone:
044-819-3121, 044-819-3111
FAX: 044-819-3120

Special Dispatch Arrangements: Provide Dispatch with the Contract number and Serial Number.
Arrangements for Non-Contract Customers: Please provide a valid PO number for billing on a Time and Materials (T&M) basis.

New Zealand

Report bugs to the New Zealand Answer Center at the number shown below.

Hotline Telephone: 4-499-2344

Your STB Subscription

Your company's software service contract includes a monthly issue of the STB. Each month, the copy of your STB is mailed to your company's primary contact person or department named in the contract. Sites with more than one contract may receive more than one STB copy, depending on how the contracts are arranged.

Duplicating Your STB

Your primary contact person or department may duplicate this "master" STB copy for all Sun workstation end-users. As long as you duplicate copies and route them internally only, there are no copyright infringement problems.

This limited permission for duplication is for your convenience only, however, and does not include any duplication for resale, for distribution outside your company, or for distribution to employees of companies not having a Sun software service contract.

Multiple STB Subscriptions

You can now order extra copies of the STB for a low monthly fee for each additional copy. These extra copies will sent to the same address as your "master" STB copy. Please contact your local Sales/Service Office or Sun Authorized Distributor.

Changes or Updates

Changes of address or other updates, including change of contact name, should be submitted to your Customer Service Administrator (CSA), not to the STB Editor. Your CSA will register the requested changes through proper channels with Sun headquarters.

Further STB Questions

If you have any questions or comments regarding the STB, including corrections, please send your comments to *stb-editor@Sun.COM* or *sun!stb-editor*.

Index

C

C, 3-15, 3-19 to 3-20, 3-21, 3-31
    C++, 3-5, 3-13, 3-21, 3-31, 3-33, 3-34 to 3-39

M

Memory Management Unit (MMU), 2-25
multithreading
    dispatch priority, 2-4, 2-5, 2-7, 2-8, 2-10, 2-16
    dispatch queue, 2-3, 2-5, 2-7, 2-8, 2-9 to 2-10, 2-12, 2-18
    global priority, 2-5, 2-7, 2-16, 2-18
    inherited priority, 2-5, 2-16, 2-18
    interrupts, 2-5, 2-6, 2-9, 2-10, 2-11, 2-21, 2-25
    priority inheritance, 2-5, 2-15 to 2-20, 2-21 to 2-22
    priority inversion, 2-2, 2-12 to 2-16
    scheduling classes, 2-6 to 2-7
    sleep queue, 2-5, 2-7, 2-17, 2-18, 2-19 to 2-20
    synchronization, 2-4 to 2-5, 2-7, 2-9, 2-14 to 2-20, 2-21 to 2-22
        condition variable, 2-4, 2-13, 2-22
        counting semaphore, 2-4, 2-22, 2-25
        mutual exclusion (mutex) lock, 2-4, 2-14 to 2-15, 2-16, 2-17, 2-21 to 2-22
        readers/writer lock, 2-4, 2-14, 2-16, 2-17, 2-20, 2-22
    turnstiles, 2-19 to 2-20
    user priority, 2-5

O

OpenWindows, 3-1, 3-2, 3-10, 3-14, 3-16, 3-18, 3-25

P

POSIX, 2-25

R

realtime scheduling, 2-1 to 2-26
    dispatch latency, 2-2, 2-8, 2-22 to 2-24, 2-25
    preemption, 2-2 to 2-3, 2-7, 2-9 to 2-10, 2-11 to 2-12, 2-15, 2-24

S

Source Code Control System (SCCS), 3-16, 3-25
SPARCompilers, 3-3, 3-12, 3-23

SPARCworks, 3-1 to 3-39
Analyzer, 3-1, 3-3, 3-4 to 3-5, 3-12
Debugger, 3-1, 3-3, 3-4, 3-5, 3-12, 3-18, 3-24, 3-32, 3-38
FileMerge, 3-1, 3-4
MakeTool, 3-1, 3-3
SourceBrowser, 3-1, 3-3, 3-4, 3-5, 3-12 to 3-39
SPARCworks Manager, 3-1, 3-3, 3-6 to 3-11
STREAMS, 2-4, 2-12, 2-13, 2-25
SunOS
multithreaded kernel, 2-3 to 2-22
System V Interface Definition (SVID), 2-2

T

Text Editor, 3-16, 3-25
ToolTalk, 3-10, 3-11
Transaction Lookaside Buffer (TLB), 2-25