Real-Time Thread Package

Kostadis Roussos

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-96-18
May 1996

Real-Time Thread Package

Kostadis Roussos
Brown University
Honors Thesis
Under the Direction of
Thomas W. Doeppner Jr.

May 20, 1996

Abstract

Solaris threads and the real-time facilities of the OS are a powerful combination for solving concurrency problems whose correctness depends on timeliness of execution. Unfortunately the RT facilities and Solaris threads cannot be easily combined by application programmers. Furthermore, the constraints on access to the RT facilities limits the potential user base of applications that make use of RT threads. The Real Time Thread Package (RTTP) is a C++ framework, that extends the Brown Thread Package (BTP), and provides a simple mechanism for writing applications that use real-time threads. This paper describes in detail the design and implementation providing both motivation and justification. The paper also discusses an application that used the RTTP in conjunction with a C++ framework for writing Producer-Consumer style programs.

1 Introduction

1.1 Operational Overview

The purpose of the Real Time Thread Package (RTTP) is to make it easier for application developers using C++ and application users to use the RT facilities, of Solaris in particular, and OS's in general. To achieve that goal the RTTP addresses a number of issues that make the Solaris RT scheduler inaccessible, difficult to use, and non-portable. The RTTP relaxes the restrictions on the usage of the RT scheduler and makes it easier and safer to use. The RTTP simplifies the interoperability of standard user-level Solaris time shared threads (TST) and real-time threads (RTT) by providing a consistent interface allowing TST and RTT to be treated uniformly by code. Finally, the RTTP provides a level of abstraction allowing application programmers to develop on a OS independent layer of code.

RTTP extends the Brown Thread Package (BTP), a C++ framework that treats threads and thread synchronization primitives as C++ types. [Jr] The BTP thread, for example, is an object whose methods provide the same facilities that Solaris and POSIX C-functions provide for manipulating vanilla threads. The BTP types are wrappers over the Solaris thread package types, thus the instantiation of a BTP type does not imply the simultaneous creation of the Solaris object. The BTP is portable and can be implemented on different OSs or using different OS vendor's thread packages without modifying the standard interface, since the implementation details are hidden. An example of such a port is the OSF-DCE version of the BTP. The BTP was chosen since it provides a portable interface to threads, and because by modeling the thread types through inheritance, code that uses the BTP Threads can also use the RTTP threads.

Widespread usage of the Solaris RT facilities is hindered by the requirement that the application run as super user. The RTTP works around this restriction and provides a more flexible type of permission restrictions. The RT scheduler provides application developers for applications that have critical timing issues a valuable tool. For example, events that need to be responded to immediately can be addressed by using high-priority threads. Additionally, continuous events that must continue to execute at a fixed rate in spite of machine load can continue to do so provided their real-time priority is high enough. Although there are uses in general-purpose applications for RT threads, their use is constrained by the Solaris requirement that the application execute as super user or the user be super user. Neither option is particularly desirable. The first because potential application users may not want to run an arbitrary program as super user, concerned over possible bugs or features in the application that make it

a security hole. The second because it makes protecting sensitive information and system integrity in UNIX impossible. The RTTP addresses the issue of permissions by providing a work around the Solaris' restrictions and, in turn, restricts access to the RT facilities by making a special user command, essential for use of the RTTP, accessible only to members of a UNIX protection group.

RT threads, in addition to the problems associated with thread development – deadlocks, contention, etc – introduce the additional danger of locking an entire node by an errant RT process. A simple example is a thread that sits in while(1) loop. If no thread with a higher priority exists that can terminate the errant process, the RT thread will make the node unuseable by preventing all processes from running and the kernel will no longer receive keyboard input making termination of the process impossible. As a result terminating the process involves power-cycling the machine. The RTTP package deals with this problem by terminating runaway processes.

There are or will be a large body of threaded programs and programmers who have had experience using threads. RTTP tries to leverage both that installed base and expertise by extending the notion of thread to threads that have scheduling parameters. The scheduling parameters of a thread are its priority and its time quantum. The current Solaris support for real-time threads makes writing an application that uses both standard threads and RT threads difficult, because to the OS there are no real-time threads, but only real-time kernel threads. As a result there is a difference between the types of objects the application programmer uses to write threaded programs and the real-time thread object. In addition, although Solaris goes to great lengths at building abstractions over its thread implementation, the use of RT threads requires, not only a thorough understanding of the model, but also an understanding of how the various parts of the model interact. The RTTP addresses these issues by treating time-shared threads and RT threads as C++ objects that are interchangeable, and have the same interface. As a result applications that require usage of the RT facilities can safely ignore the underlying layers of the OS and develop at the highest layer of abstraction.

The Solaris RT support is not portable. Its interface is very tightly coupled to the underlying OS reducing application portability. Application programmers that want to use RT facilities of a RT OS in a portable manner must build layers of abstraction over Solaris. The RTTP provides such a portable layer.

The RTTP is an improvement over the direct use of the Solaris RT facilities because it relaxes the requirements for usage, it is safer to use, it simplifies incremental development of RT threaded applications, and it successfully encapsulates the underlying thread model in a portable way.

1.2 Architectural Overview

RTTP consists of a set of private and public objects that provide entry points to all the facilities of the Solaris RT scheduler as extensions to the BTP. The design philosophy was to create a set of classes that extended the BTP but added as few new public types as possible. In addition, the interface was designed to hide the underlying Solaris thread model so as to simplify usage and enhance portability.

RTTP provides only four new user types. They are *RTThread*, derived from *Thread*, *RTJoinableThread*, derived from *JoinableThread* and *RTThread*, *RTParams*, derived from *ThreadAttr* class, and *RTManager*. *RTThread* and *RTJoinableThread* are used as base classes for any RT thread objects. They differ from their time-shared counterparts in that they have facilities for setting their scheduling parameters and run in the real-time scheduling class. Implicit in the value of the priority is the scheduling class of the thread since certain values can only be used by RT threads. *RTManager* provides an interface to the scheduler that is simpler and cleaner to use and obscures the Solaris interface. *RTManager* also has facilities for changing thread parameters, and for querying about thread parameters.

A design goal of the BTP was to provide an OS independent C++ layer similar to the OS independent POSIX interface to threads. The RTTP maintains that philosophy. The OS dependent components are hidden in the private sections of the C++ class hierarchy and by the *RTParams* data type. The net effect is that RTTP is orthogonal to the Solaris RT scheduler and Solaris threads. It is orthogonal to the Solaris RT scheduler because the OS dependent parts are not publicly accessible. RTTP is orthogonal to the Solaris thread package because it is built on a portable wrapper on top of the Solaris thread package.

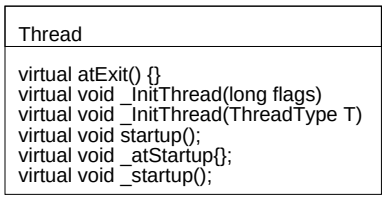

Figure 1: Extensions to Thread Class

2 Implementation Details

2.1 Overview

From a technical point of view, there were a number of obstacles to the implementation of the Real Time Thread Package (RTTP). The simplest was how to build real-time (RT) threads out of the components of the Solaris thread model. Although the BTP was intended to be extensible, extending it required reworking the *Thread* class. A number of methods were added as a well as a change in the initialization of the underlying Solaris thread. The most complex problem was the development of a means for working around the access restrictions imposed by Solaris. The third problem was the development of the dead-man switch, a software switch that prevents runaway real-time threads from hanging the machine. Finally, the last technical hurdle was the correct initialization of the RTTP.

The RTTP could not directly schedule Solaris user-level threads as real-time threads because of the Solaris thread model which is a hybrid model made up of two distinct layers: a kernel level scheduler that schedules kernel threads, called lightweight processes (LWP), on CPUs; and a user-level threads library that schedules user-level threads on LWPs. The RTTP *RTThread* must be scheduled directly by the kernel because it is defined to execute for some time quantum of the global time and the user-level threads library can only schedule a thread to run for some time-quantum of the total time the process the thread is in gets to run. The desired effect of having the thread scheduled by the kernel and not the user-level library was achieved by binding the thread to an LWP. Binding the time-shared thread (TST) to the LWP, ties them together such that whenever the LWP runs, the TST runs, and no other TST may be scheduled to run on that LWP. It was then possible to use the underlying LWP to have the thread scheduled by the kernel. [Sunc] The RTTP *RTThread* had to use the Solaris thread types because there is no other convenient way to create user-level threads.

Although the BTP is a portable C++ framework, the internals of the BTP *Thread* class had to be modified (Figure 1) to get real-time threads to work. A design goal of the RTTP was to use the BTP *Thread* as the base type of the real-time thread so that real-time threads and time-shared threads could be used transparently in code. The real-time thread initialization procedure differs substantially from the initialization procedure for time-shared threads. In particular, the real-time thread is not created directly, but indirectly by another thread. Furthermore, the RTTP needs to perform certain operations before the thread begins to execute client code, but after the underlying Solaris thread has been created. The original BTP *Thread* did not provide a mechanism for either changing the mechanism of initialization or for adding hooks to be called before *Thread::startup*. *Thread::startup* is a pure virtual function that is overridden by a subtype and contains the code that is executed by the thread. As an additional constraint any modifications to the BTP could not change its public interface as there was legacy code that depended on it. Changes are invisible to client code. Originally the Solaris thread was created inside the *Thread* constructor. The Solaris library call *thr_create* was passed a static member function, *Thread::real_startup*. When it began execution it was passed the *this* pointer of the thread object as a parameter which it used to then call *Thread::startup*. Now, rather than have *thr_create* called inside the constructor, it is called inside of *Thread::_InitThread*, a virtual method, called by *Thread::make_runnable*. Having *Thread::_InitThread* called inside of a function rather than inside of a constructor makes it possible for a virtual method to create the thread thus allowing different kinds of initialization by different thread types. To provide hooks for code to be executed before *Thread::startup*, *Thread::real_startup* calls *Thread::_startup* instead of *Thread::startup*. *Thread::_startup* calls *Thread::atStartup* and then *Thread::_startup* calls *Thread::startup*. *Thread::atStartup* is a virtual function that can be overridden to execute different code for different *Thread* subtypes. The extensions to the BTP did not change the public interface of the Thread class, while at the same time providing the facilities necessary to extend the thread class.

The most complicated problem addressed by the RTTP is the means to work around requirement that to use the RT facilities of Solaris, one must be superuser. We first tried an approach that, though it had the desired functionality was slow and prone to failure. The basic idea was to use a program to change a process's scheduling class to RT. The newly created RT process's LWP would all be RT with the parameters used to convert the process to RT. The process would then convert all of the LWPs not originally RT to be TS. The approach the RTTP uses is first to create a server RT thread at start up as before, and then used it to create other RT threads.

The first approach, shown in Figure 2, is called the FORK-EXEC Approach. To

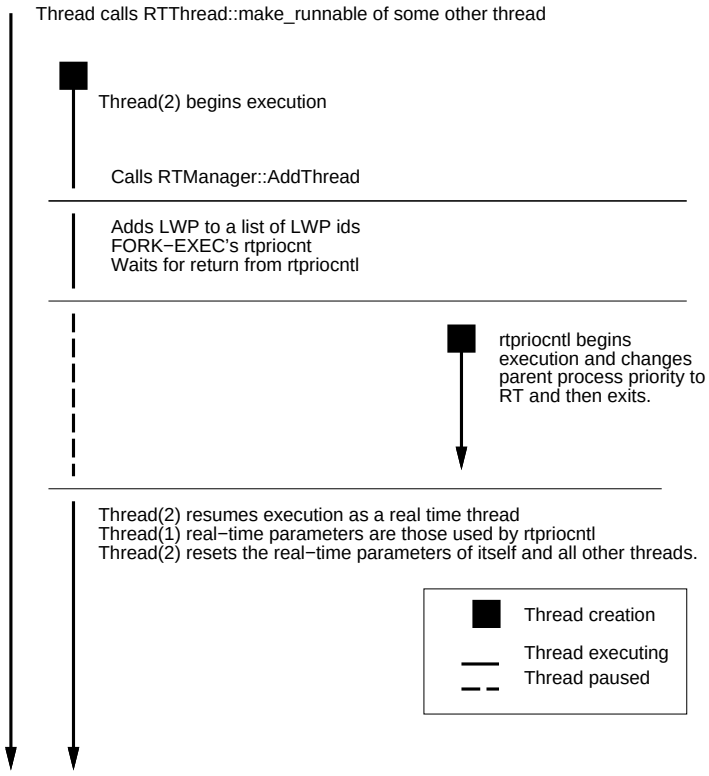

Figure 2: FORK-EXEC Approach

change a process's scheduling class to RT the process making the change must be root. Our original approach was to use the Solaris user command *priocntl* and execute it with an effective user ID of root to change the scheduling class of a process to real-time. [Sunb] Unfortunately *priocntl* changes its effective ID to real user ID before it calls the *priocntl* system call. As a result we modified the user command *priocntl*. The modified version *rtpriocntl* differs from the original version in two ways. The first is that it does not reset its effective user ID to real user ID just prior to executing the *priocntl* system call. The second difference is that it is only accessible to those in a particular UNIX protection group. Using *rtpriocntl* we were able to change the scheduling class of a process to RT conveniently and securely. As a side effect to changing the process's scheduling class the scheduling parameters of all the LWPs of the target process were set to the parameters passed to *rtpriocntl* and, therefore, had to be changed back to their original values. [Sund] The exact procedure detailed in Figure 2 was used to make real-time threads using *rtpriocntl*. First some thread creates a thread object and then calls *Thread::make_runnable* to start the underlying Solaris thread. *Thread::make_runnable* then calls *RTThread::_InitThread* that in turn calls *RTManager::AddThread*. *RTManager::AddThread* first adds the

LWP ID of the calling thread to a list of RT LWPs. *RTManager::AddThread*, then *fork*s and *exec*s *rtpriocntl*. The LWP ID is added to a list because after *rtpriocntl* has finished executing the original scheduling parameters will have to be reset. The calling thread then waits for the modified *priocntl* to return. The thread then uses the list of LWP IDs to change all non RT LWPs to TS and reset the parameters of any RT thread. When it finishes resetting all the LWPs, it returns from *RTManager::AddThread* and continues execution. The problems with this approach are both obvious and subtle. The biggest disadvantage is that the creation of an RT thread becomes very expensive as creating a new thread involves a *fork* and an *exec*. Another obvious disadvantage is that every time a *RTThread* is created, all the priorities of all the threads are reset. As a result some very important RT threads during the period of thread creation, may not get a chance to run for a critical interval. A slightly subtler complication is that the creation of the *RTThread* requires the execution of a time-shared process. Depending on the real-time load of the system, *rtpriocntl* may never return, effectively hanging a thread.

A partial solution to the problems associated with the FORK-EXEC approach was to use the FORK-EXEC approach to create a server, or "mother" thread, and then use that thread to create other RT threads (see Figure 3). The new approach is based on the observation that an LWP when created has the scheduling parameters of the creating LWP. Although it is possible for any real-time LWP to create a new real-time LWP, the mechanism used by the RTTP is to have a single real-time thread, the "mother" thread, create all other real-time threads. The existence of *G_mother* and the mechanism for thread creation (detailed in Figure 3) can be ignored by the user for the most part. The global instance variable *G_mother* of type *RTMother* is used in the RTTP system to represent "mother". *G_mother* is made real-time by using the FORK-EXEC Approach and after initialization sits in a while(1) loop waiting for requests to create a new thread. When a create thread request is made by calling *G_mother->GiveBirth*, *G_mother* runs, pre-empting any other running thread and creates a new real-time thread. The new approach is an improvement over the original approach, but not an ideal solution. The fundamental advantage, of the new approach, is that creation of a new thread is not as expensive an operation. Whereas before, an entire new process had to be created, and the calling process was suspended until *rtpriocntl* returned, the new approach is almost as quick as a function call. The new method, however, can still suspend a thread if there are insufficient processors. When "mother" executes, no other thread can execute on a particular processor since it runs at the highest priority, potentially suspending a critical thread. The new approach is, nonetheless, better than the old approach which suspended the entire process until another process terminated and then had to reset all the parameters of all the LWPs. Although during execution of a process the creation of threads is faster, a FORK-EXEC is still required at system initialization, so start up of the RTTP is still slow. Finally, the new approach consumes system resources, in particular a new thread must stay around

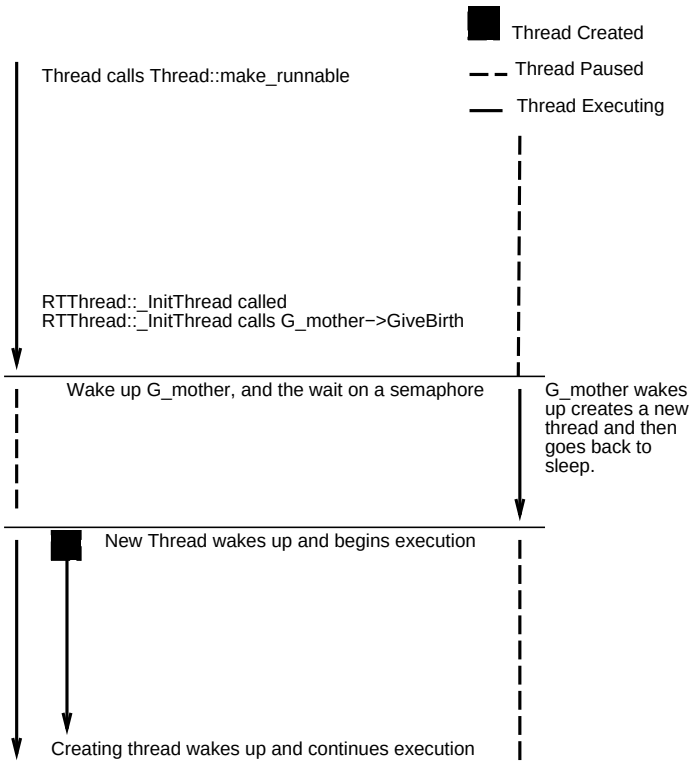

Figure 3: Creating a Thread Using a Mother

and the priority of *G_mother* is now no longer accessible to developers. The approach taken to creating RT threads without requiring super-user privilege by the program using the RT scheduler allows arbitrary users who have access to *rtpriocntl* to use the RT scheduler. The approach is efficient and mostly transparent to the user, but nonetheless suffers from some minor flaws.

Preventing lock up by a runaway RT thread was an interesting technical problem. Lock up occurs when an RT thread does not relinquish the CPU preventing all other LWPs from executing. To stop an errant RT thread the system must be rebooted. This is unacceptable. The approach taken by this package is to use two additional threads to monitor whether the system has locked up and if so terminate the program. [HJT+93] The two threads are objects with type names *DeadManThread* and *DeadManSwitch*. The *DeadManThread* is a normal thread. Upon creation it sets a bit, and then goes to sleep for a period of time wakes up and sets the bit again. The *DeadManSwitch* is a real-time thread. Upon creation it goes to sleep for a period of time such that the *DeadManThread* has had sufficient opportunity to run. When it wakes up it checks the bit and if the bit is not set it terminates the process. The limitations of this approach are

significant. The most serious is that the dead-man switch may turn on, despite the fact that forward progress is being made by the program. As a result, the use of a dead-man switch or not is a compile time option. A second limitation is that if the code executed by a user real-time is the following:

```
MYThread::startup {

       while(1);
}
```

then the dead-man switch will not work. Although it would appear that the this is exactly the type of program that the dead-man switch was designed to protect against and therefore does not work, it turns out if the code inside the thread is more substantial than just a jump to the same address, then the dead-man switch works. For example the following thread is terminated by the dead-man switch:

```
MYThread::startup {
 while(1){
         k++;
         l++;
         m++;
         if((k > 100000) && (l > 100000) && (m > 100000)){
                x++;
                if(x == 100000000)
                     break;
         }
 }
```

The second code snippet has more instructions inside of the loop that is executed by the real-time thread. It is unclear what the difference between the two threads is, other than the length of code. In spite of this peculiar limitation, the dead-man switch proves to be a very useful addition, reducing machine reboots.

The correct order of initialization of the system components of the RTTP was a technical problem requiring some care. The initialization of the RTTP has two phases (see Figure 4). The first is the initialization of the dead-man switch, and the second the creation of *G_mother*, the singleton instance of *RTMother* or the mother thread. *G_mother* must begin accepting requests for creating

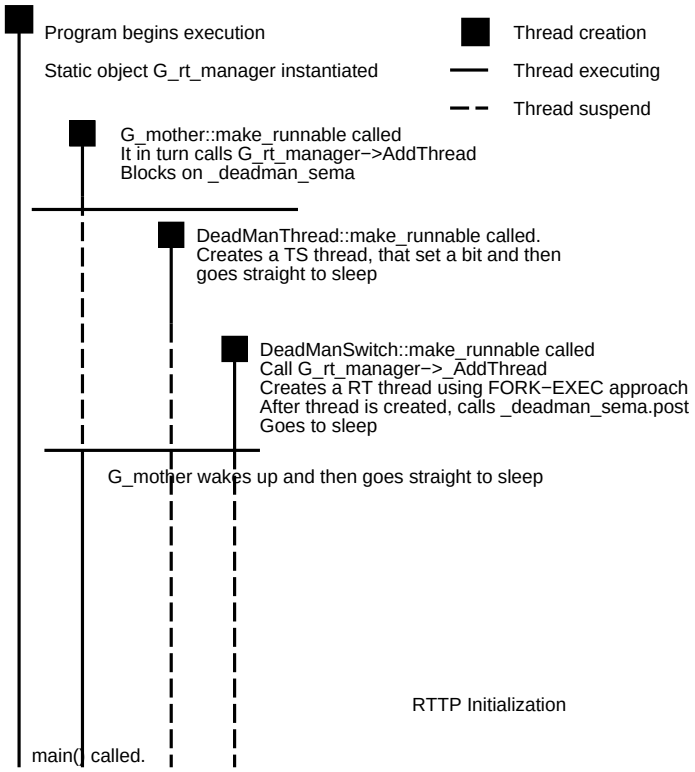

Figure 4: Initialization Sequence of RTTP

new threads only after the dead-man switch is set up. There is a potential race condition if no synchronization is used. If a runaway RT thread is created by *G_mother* before the dead-man switch is operational, the dead-man switch will never start since the creator of the dead-man switch is a TST. As a result, *G_mother* cannot be used to create the *DeadManSwitch*. The procedure for initialization is first to create the *DeadManSwitch* thread and call *DeadManSwitch::make_runnable*. That in turn calls *RTManager::_AddThread* that uses the FORK-EXEC approach to make the *DeadManSwitch* RT. After the *DeadManSwitch* has finished its start up, it posts the semaphore that the *G_mother* waits on, and then *G_mother* uses the FORK-EXEC approach to go real-time as well. After mother is alive, any thread can be made RT. An important note here is that all of this takes place before *main()*. As a result, no additional work needs to be done by the user inside of *main()*. The RTTP's initialization is seemingly simple yet there are certain synchronization subtleties that must be dealt with.

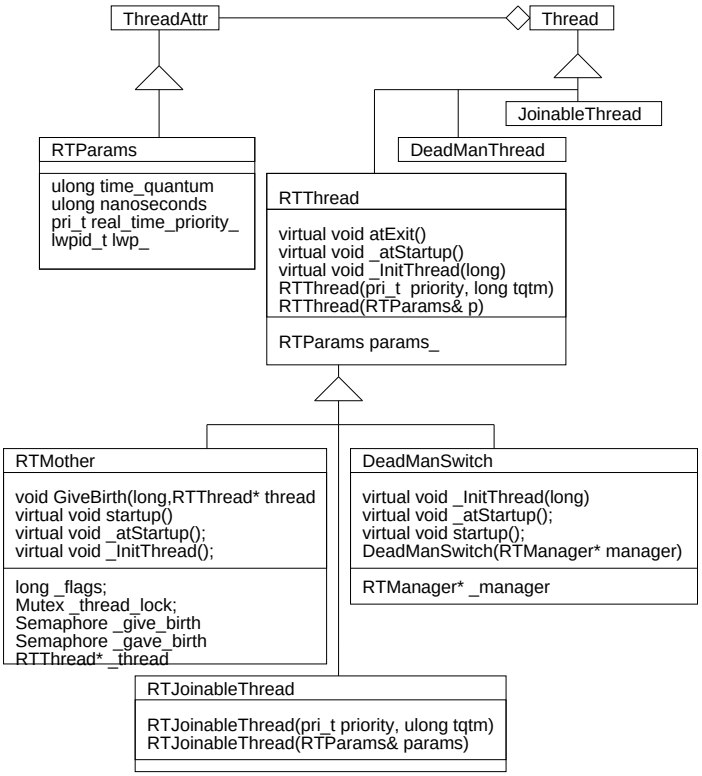

Figure 5: OMT Diagram of Class Hierarchy

2.2 Components

The RTTP is an extension to the BTP. The RTTP adds the public *RTThread*, *RTThreadJoinable*, *RTManager* and *RTParams* classes as well as the private *DeadManSwitch*, *DeadManThread*, *RTMother*, and *FirstObject* classes (Figure 5 shows the relationship of the classes to one another). In addition, the RTTP adds *G_mother* and *G_manager* as global objects that are created before *main()* and are destroyed after *main()* exits.

The *RTThread* and *RTJoinableThread* are the base types for all real-time threads (Figure 5) . *RTJoinableThread* is a convenience type inheriting from *Joinable* and *RTThread* providing no new functionality. To create a new thread, the user is expected to derive a type from either real-time thread type and override the *Thread::startup* method, which is the code that the Solaris thread executes. The differences between *RTThread* and BTP *Thread* are isolated to internal thread start up and termination, and do not impact the user. *RTThread* over-

rides *Thread::_InitThread()*, *Thread::_atStartup* and *Thread::atExit*. Whereas *Thread::_InitTread* creates the Solaris thread directly, *RTThread::_InitThread* asks *G_mother* to create the Solaris thread. When the Solaris thread is created it eventually calls *Thread::_startup* that in turn calls *Thread::atStartup*. *Thread::atStartup* does nothing, but *RTThread::atStartup*, which is called when a real-time thread is created, changes the LWP's scheduling parameters from those of *G_mother* to those in *RTThread::params_*. The scheduling parameters are set in *RTThread::_atStartup* and not by *G_mother* because to set the parameters you need the LWP ID of the underlying thread, which can only be acquired by calling *_lwp_self()* while the thread whose bound LWP we are interested in is executing. After the parameters have been reset, the derived type's *startup* method is called. To terminate a thread, *Thread::exit* is called. that in turn calls the virtual function *Thread::exit* whose default behavior is to do nothing. *RTThread::exit*, on the other hand, removes the thread from the RT scheduling class. After it returns, the Solaris library function *thr_exit* is called and the thread terminates.

RTMother is the type for the "mother" class that is used to create other real-time threads. To create a real-time thread, every *RTThread* subtype effectively makes a remote procedure call to the "mother" thread requesting thread creation. The "mother" thread is initialized using the FORK-EXEC Approach and after initialization executes the following code:

```
void RTMother::startup()
{
        while (1) {
                _give_birth.wait();
                // Here the Solaris thread is created
                // _thread is a pointer that points to the RTThread
                // that called RTMother::GiveBirth
                _thread->InitThread(_flags);
                _gave_birth.post();
        }
}
```

Immediately upon entering *startup*, *G_mother* is suspended waiting for a thread to ask to create it. It is woken up when any *RTThread* calls *RTMother::GiveBirth*.

```
void RTMother::GiveBirth(long flags,
                         RTThread* thread)
{
        _thread_lock.lock();
        _flags = flags;
        _thread = thread;
        _give_birth.post();
        _gave_birth.wait();
        _thread_lock.unlock();
}
```

The calling *RTThread* first wakes up *G_mother*, and then waits for *G_mother* to finish creating the underlying Solaris thread. Note, that threads are serialized into *RTMother::GiveBirth* because *G_mother* is single-threaded and does not use a concurrent FIFO data structure that would allow a concurrent enqueue and dequeue of thread create requests. Using a concurrent data structure did not seem appropriate given the size of such data structures, and the unlikelihood that this would be a point of high contention, the assumption being that real-time threads are not created often and concurrently.

The *DeadManSwitch* and *DeadManThread* are used to avoid deadlock. They exist to prevent a real-time thread from starving out every other thread of execution by never relinquishing the CPU. The principal behind this approach is to run a real-time thread at the highest possible priority and have it periodically check whether a thread of time-shared priority has run and if it has not to terminate the process. The *DeadManThread* is a a vanilla BTP *Thread* object that runs as time-shared thread. The *DeadManThread* only overrides *Thread::startup*. Inside of *DeadManThread::startup*, *G_rt_manager->setBit* is called, and then thread goes to sleep repeating forever. *DeadManSwitch* runs at the highest system priority. It inherits from *RTThread* and overrides *Thread::startup*. *DeadManSwitch::startup* first goes to sleep and then wakes up after a period of time, at which point it calls *G_rt_manager-> testBit*. *RTManager::testBit* unsets a bit and returns the old value. If the bit is set, then the thread goes back to sleep, otherwise it exits the process, since that means that the *DeadManThread* never got a chance to execute.

The *RTManager* (Figure 6) provides a portable interface to the RT scheduler, and manages the initialization of the RTTP. The facilities provided by the *RTManager* are accessed through the singleton instance of the *RTManager*, *G_rt_manager*. The facilities for modifying or querying LWPs are basically wrappers over the *priocntl* system call. *RTManager* provides facilities to remove threads from the RT scheduling class and put them in the TS band, to change the

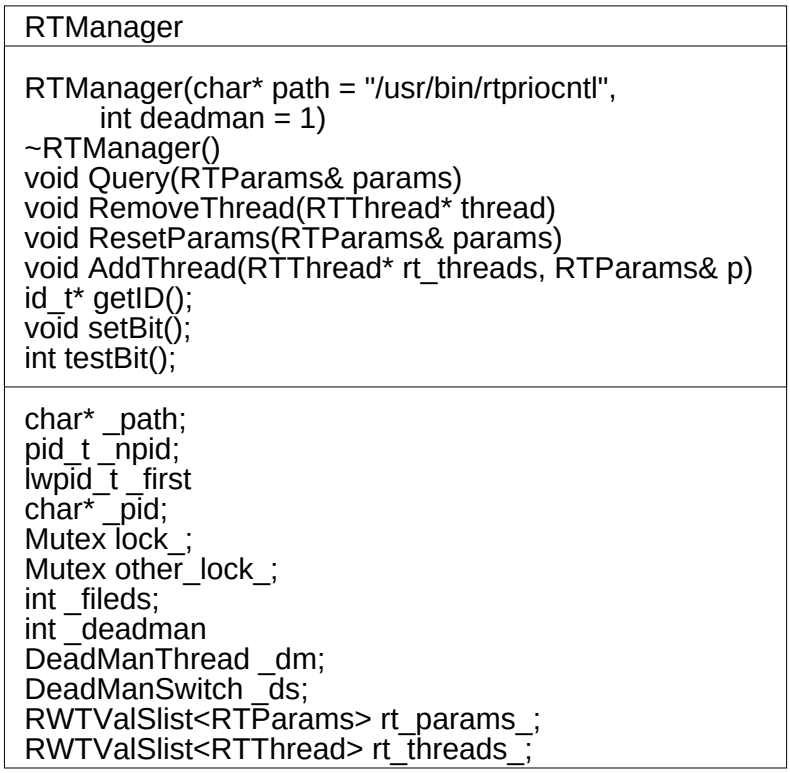

Figure 6: RTManager

real-time parameters of a thread, and to get the *RTParams* of a thread. The initialization of the RTTP involves initializing a global object, *G_rt_manager* and the dead-man switch all of which occurs before *main()* so that no additional work is required when the program begins execution. *G_rt_manager* is initialized using a variation on the mechanism used by Stroustrop to initialize *cout*, *cerr* and *cin*. Essentially a static object is instantiated and as part of its instantiation it creates an instance of *RTManager*. *RTManager* on instantiation first checks to see if the environment variable PRIOLOC is set, indicating a location different from the default one for *rtpriocntl*. When initializing "mother" and the dead-man switch, it must guarantee that "mother" is created after the dead-man switch. If no synchronization takes place there is a potential race condition between an errant RT thread and the creation of the dead-man switch. If the errant process begins execution before the dead-man switch which is created by a time share thread, then the dead-man switch will never be started, and the RT thread will lock the node. After initializing internal data structures, *RTManager* calls *G_mother->make_runnable*, and initializes the dead-man switch. *RTManager::AddThread* and *RTManager::_AddThread* are used to create a thread using the FORK-EXEC procedure. *RTManager::AddThread* is used by *RTMother*

and *RTManager::_AddThread* by the *DeadManSwitch*. *RTManager::AddThread* contains a semaphore that the callee must wait on until the *DeadManSwitch* terminates its initialization and then calls *RTManager::_AddThread*. *RTManager::_AddThread* adds the callee to a list of *RTThread*, called *rt_threads_*, and then calls *fork1*. The child then immediately forks *rtpriocntl*. The parent waits for the child to terminate, if the child does not terminate successfully, then the parent throws an exception. If the child terminates successfully, all the LWPs of the parent process are real-time LWPs whose priorities are equal to whatever real-time values *rtpriocntl* set the parent process to. The thread that originally called *RTManager::_AddThread* then sets all of the LWP not in *rt_threads_* to be TS and resets the parameters of the LWPs in *rt_threads_* to their original value and then resumes execution.

3 The Solaris Interface

The Solaris interface to the real-time facilities of the OS and the support for real-time threads is problematic. The interface between the user-level thread scheduler and the kernel scheduler is not straightforward. The kernel scheduler requires that the user of the RT facilities have an effective user ID of root. Finally the facilities provided for RT scheduling are not complete.

The kernel scheduler of LWPs and the user-level thread scheduler work together to schedule threads. The kernel scheduler, however, is ignorant of the user-level scheduler. The Solaris thread package is designed to allow the developer to ignore the underlying LWP and thread model. It does allow for the user to provide hints to it and the underlying scheduler, but there are no means for control. To set kernel scheduling parameters you need to bind a thread to an LWP so that scheduling the LWP is equivalent to scheduling the user thread. The major problem with this approach is that it breaks the encapsulation, since an application must use non-portable OS specific code to initialize the real-time threads, and keep data that is specific to the particular Solaris implementation to manipulate the threads. This is an issue because programmers must now expose their code to the underlying non-portable thread model, which may change. Furthermore, the purpose of the Solaris thread package is to encapsulate the Solaris thread model.

The requirement that the effective user ID of programs that use the real-time facilities be root is extremely restrictive. Although there is a means for working around the *priocntl* system call's restriction, *mlock* and *munlock*, the memory locking facilities resist such an approach for locking arbitrary pages of memory. Memory locking is extremely important because a real-time thread must be able to guarantee continuous memory residence to prevent paging and swapping [Sund]. If the memory of a thread is not in memory, the thread will be blocked and descheduled in spite of its scheduling parameters. Unfortunately a process can only lock its own pages and to do so must be root. Because another process cannot lock another process's pages, the approach used to get around the restrictions on *priocntl* is not useable. Although the restrictions on using the real-time scheduler and locking memory are understandable given their potential effects on performance, this effectively eliminates the usage of the RT facilities from the main stream as developers have a very restricted audience.

4 Multi-Threaded MPEG Viewer

An application that benefits from the use of real-time (RT) threads is a multi-threaded (MT) MPEG Viewer originally developed independently of the Real-Time Thread Package. MPEG is a compression standard devised by ISO, used to compress digital video [Gad]. The MT MPEG Viewer essentially decodes an MPEG version of a movie clip and then displays it to screen. The MT MPEG Viewer is built on top of a C++ class framework for concurrently decoding and displaying MPEG movies. The MT MPEG Viewer concurrently reads in a number of MPEGs from disk, concurrently decodes the next frame and then displays them. Rather than display frames as you get them, such that a simple movie to decode would display at a faster frame rate than a more complex movie, the MT MPEG Viewer displays at the frame rate of the slowest MPEG to decode. The C++ framework consists of a reentrant version of the Berkeley MPEG decoder, developed at Brown University, and a display module that manages creating, and displaying the data to window using X11. The application decodes the films in parallel, enqueueing each new frame on separate queues, while a single thread dequeues a single frame from each queue.

The MPEG viewer was originally written using time-shared threads, but suffered from jitteriness and load-effects and was modified to take advantage of the real-time threads package. It suffered from jitteriness because either the thread decoding the film or the thread displaying the frames would be descheduled for a period of time so that some other processes could run. As a result there would be occasional pauses. By using real-time threads, both the consumer and the producer were guaranteed to run continually and not pause. The program also suffered from load-effects. As the load increased on the system by some other user starting a compile for example, the frame rate dropped only to pick up when the compile terminated and the load decreased. Using real-time threads guaranteed that the threads always were able to execute regardless of the time-shared load of the system. The program flow of control consists of a number of stages. The first stage decodes the next frame and enqueues the frame. For each film there is a different queue. The second stage dequeues a frame from each queue and then displays the frame. The first stage consists of a series of producer threads, that produce frames, and the second stage consists of a single thread that consumes the frames produced. Both the first-stage and second-stage threads are real-time threads.

The multi-threaded MPEG viewer uses a C++ framework for writing producer-consumer style programs – programs in which there is both a source of and a sink of data and (Figure 7). This problem can be boiled down to writing a multi-threaded FIFO data structure. A program that has a source producing data at irregular intervals and has a sink that consumes data can be re-formulated

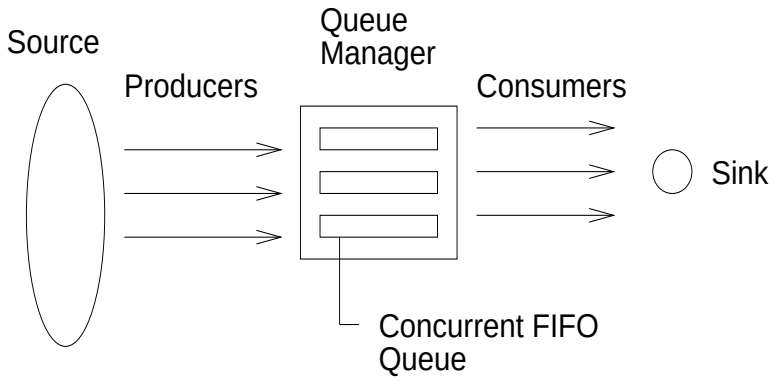

Figure 7: Generic Producer-Consumer Framework

as producer-consumer program. In particular this includes programs that must acquire data from a device, and process the data. If the data is produced at a much faster rate than it is processed, having a single thread both read the data from the device and process it will result in loss of data from the device, especially if the device has a bounded buffer to store data it generates. A better approach is to use a simple thread to obtain the data from the device, enqueue the data and then wait for the next piece of data from the device, and have other threads dequeue the data and process it. The framework was developed independently of the real-time threads package. The C++ framework consists of a set of classes that abstract away the synchronization and the underlying FIFO data structure. The user of the framework needs to only define the mechanism for producing and consuming data and string the producers, consumers and FIFO data structures to build the desired pipeline.

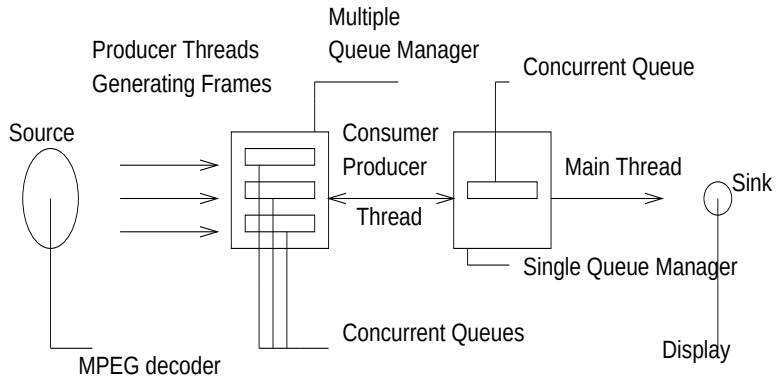

Figure 8: Multi-threaded MPEG Viewer Pipeline

The application itself consisted of two pipelines with three stages (Figure 8).

The third stage was a feature of the support software and not the application. The first stage consisted of a single real-time producer thread per MPEG that decoded the next frame of the MPEG. Making the producers real-time threads meant that the rate of production was unaffected by load from time shared processes. The frames were then enqueued, on separate queues. The second stage consisted of a consumer that read a single frame from each queue and then enqueued the frame. This stage ran at a real-time priority to prevent the producer queue from growing without bound. The second stage could not directly display because X11 requires that the thread that actually writes the data to socket that the server reads from be the main thread. The third stage was the main thread. It read from the second queue, and wrote the data to the screen. The decoded frames were serialized so that movies were displayed at the same rates. The third stage ran at a time-shared priority since the first thread can not be bound to a real-time thread since there is no way to get its thread ID.

The C++ framework simplifies the development of producer-consumer applications by providing classes that manage the synchronization of the pipeline. The basic components of a producer-consumer problem are the thread types, the queue manager and the queue. The type of pipe, both its operation and the underlying FIFO queue can be dynamically altered. The base type for all producer-consumer threads is *PCThread*. It provides some default functionality needed by the framework. A consumer thread must inherit from both *PCConsumer* and *PCThread*. *PCConsumer* provides the additional functionality needed to be a consumer. Since the consumer may not necessarily be a thread, the consumer facilities must be independent of the threading. The default type for consumer threads is *PCConsThread*, and producers *PCProdThread*. The default type for real-time consumers is *PCRTConsThread* and similarly for producers *PCRTProdThread*. The user needs to only define the *Consume* or *Produce* method. The rest is handled by the framework. The queue manager controls access to the underlying queue data structure. For example the first pipe of the application is an ordered set of tuples of queues and producer threads. The consumer iterates over the set, dequeueing an element from each queue. The C++ framework provides a class, named *PCMpegThread*, that manages both the producers, queues, and the consumer. The data structure used to model a concurrent FIFO queue is independent of *PCMpegThread*. In this application it is a concurrent queue, named *PCSpinBuffer*, that permits a single enqueuer and a single dequeuer to operate simultaneously, but allows concurrent access by both the enqueuer and dequeuer if the size of the queue is greater than one. The choice is appropriate for this application since there is only one producer and only one consumer per queue. The enter producer-consumer is in turn managed by *PCBuffer*. The use of the C++ framework for producer-consumer MT programs encapsulates the underlying synchronization providing the user a simpler interface.

The migration from an MT MPEG Viewer that uses TS threads to one that uses RT threads demonstrated that *RTThread* is extensible, that RT threads and TS threads can be used interchangeably, that replacing TS threads and RT threads requires some care and that changes to design can be isolated in the *RTThread* class, and that real-time threads can be used to reduce the variance in throughput due to load. The producer-consumer framework used to develop the MT MPEG Viewer was developed before the RTTP. The framework deals with threads which are subtypes of *PCProducerThread* or *PCConsumerThread*. Adding the new thread type was a trivial extension to the framework base types:

```
// The producer-consumer framework real-time consumer thread.  template
<class T> class PCRTConsumerThread : public PCConsumerThread<T>,
public RTThread {
public:
        PCRTConsumerThread(pri_t priority,
                    PCBufferImplementation<T>* buf_impl,

ulong tqtm = 100)
                :PCConsumerThread<T>(buf_impl),
                 RTThread(priority, tqtm),
                 Thread(Thread::joinable, Thread::ThreadAttr(1))
        {;}
public:
        virtual void startup();
};
```

Since *RTThread* is a subtype of *Thread*, converting the MT MPEG Viewer to using *RTThread* involved only changing the thread initialization to create *RTThread* instead of *Thread*. In particular, the queue manager did not have to be changed since it expected *PCConsumer* and *PCProducer* objects which were independent of the threads. The re-entrant MPEG decoder and display modules were able to use *RTThread* directly as it is a subtype of *Thread*. The implementation of the real-time versions of the *Consume* and *Produce* were different from the time-shared versions. The *RTThread* version of *Consume* and *Produce* had to stop after every frame to allow the TS processes, and in particular the X server time to execute. Choosing the optimal value or even the correct value for an application is domain specific. It is very important that RT thread release control of the CPU or else the TS threads will not execute. This in fact turns out to be the most important difference between the TS and RT versions of the MT MPEG Viewers. The changes required to use the real-time scheduler were isolated in the subtype of *RTThread* and did not litter the code. Both the fact that *RTThread* is a subtype of *Thread* and that the design changes imposed by using the RT scheduler can be isolated in the *RTThread* subtype

indicate that it is possible to first develop the program using *Thread*s and then use the *RTThread* for fine tuning.

The RT version of the MT MPEG Viewer is able to sustain throughput in the face of increasing TS load. In face, the RT version of the MT MPEG Viewer's frame rate did not decline appreciably with load (Figure 9). The load plot points were obtained using the *top* UNIX utility and the frame rates were obtained from the MPEG viewer itself. *top* is a UNIX utility that display the load of top 15 processes on a machine and the current load and average loads of the machine [Suna]. The experiments were run on a SPARC 10 with 2 processors and 80MB. The MT MPEG Viewer was run twice once using RT threads and once using TS threads. The load was increased using a simple program that started four bound threads that spun in a loop. As the load increased, the points were recorded and the associated load indicated by *top*. The purpose of the experiments was not to obtain accurate points, but to obtain a trend line of points. As the load increases, the MT MPEG Viewer that uses the vanilla BTP *Thread* has a degradation in performance as load increases, whereas as *RTThread* does not. The BTP version of the MT MPEG Viewer suffers a degradation in performance because the TS load of the system increases and the thread must compete with the increasing TS load for time to execute. The RT version of the MT MPEG Viewer does not suffer a degradation because the RT load does not increase and the TS load does not affect the amount of time an RT thread gets to execute. The RT thread always get to run when it is ready to run and no RT thread of higher priority is currently running.*RTThread* does not maintain an exact rate for the frame rate as load changes as there are components of the MT MPEG Viewer that are not running in RT, and the memory pages of the system are not locked. Nonetheless, the use of RTTP gives the developer a mechanism for ensuring performance even in the face of increasing load.

The experience with the MT MPEG Viewer application showed that a real project could be done using the RTTP and that old code could be modified to use the RTTP without substantially revising it.

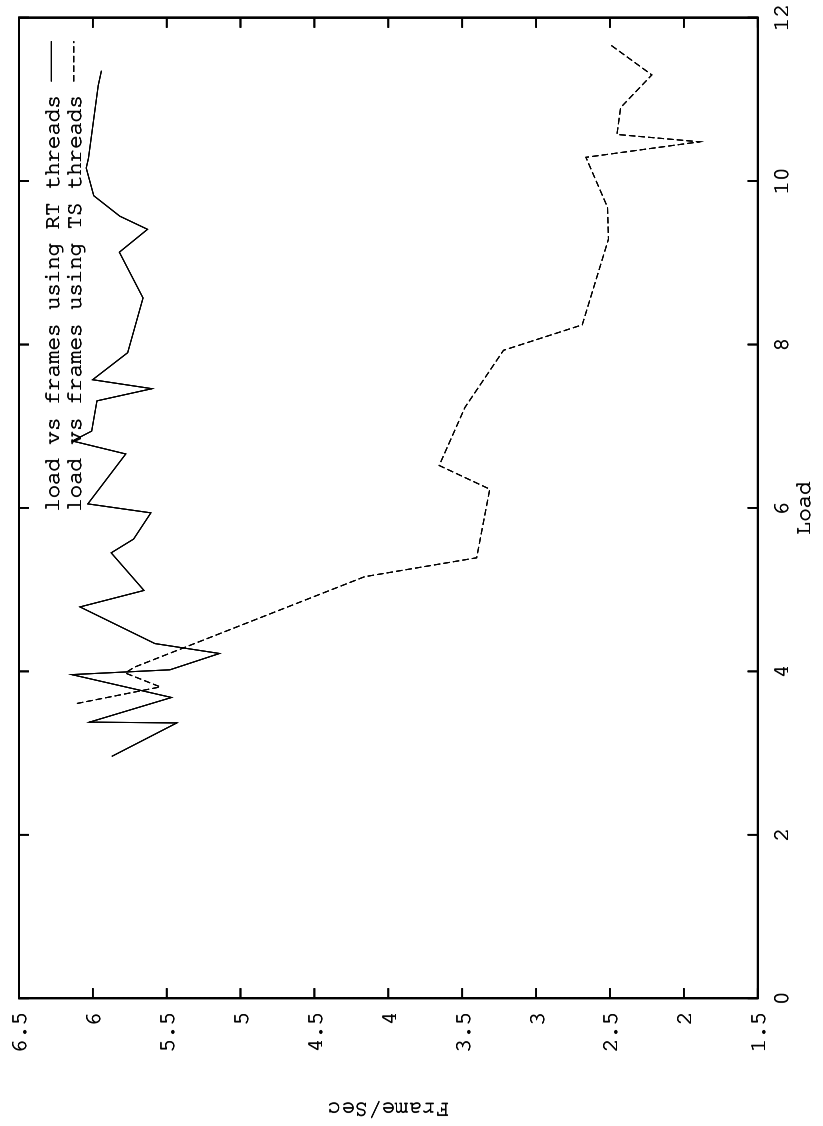

Figure 9: Load vs Frame Rate

5 Conclusion

The Real Time Thread Package (RTTP) is a viable framework for developing threaded applications that have threads that must or should use the RT scheduler. The RTTP provides a simple-to-use interface that is potentially very portable and is safe. The experience with the Solaris interface suggests that the hybrid model is problematic when parts of the OS cannot communicate effectively. The MT MPEG Viewer demonstrated that the RTTP was not only a viable package for writing new threaded programs but could be used to modify existing threaded applications that use the Brown Thread Package.

6 Appendix A: Users guide

The purpose of this appendix is to provide pointers for programmers who want to use Real Time Thread Package. The first section discusses some issues in designing applications that use both real-time threads and RTTP. The second section describes how to actually write a real-time threads program. The final section discusses compile issues.

6.1 Designing Programs that Use RTTP

When writing an application that uses real-time threads it pays to first use the BTP *Thread* to do prototyping in. An application whose program correctness depends on timeliness of execution and correct execution, is harder to debug due to the added complexity. Since the *Thread* and *RTThread* can be easily exchanged it makes sense to reduce complexity by first fixing the synchronization bugs, and then adding the *RTThread*s.

When writing applications that use the RTTP threads it is important to recognize that an *RTThread* will execute as long as it is runnable. There are important ramifications. It is not possible to directly set the amount of time a thread executes before it terminates. As a result, it is not possible to succinctly ask that a thread execute for nine tenths of a seconds and then pause for a tenth of a second. Another ramification is that if the thread does not relinquish control of the CPU and does not terminate, will cause the system to lock up if all available CPUs are being used by such threads. If the system does lock up, the only way to restart it is to reboot the machine. The dead-man switch terminates the process rather than forcing a reboot. Although using the dead-man switch does have some overhead, a process terminating gracefully is more interesting than a process that forces a system reboot.

6.2 Writing Programs that Use RTTP

There are two parts to writing a program that uses RTTP. The first is writing a real-time thread, and the second part is accessing the *RTManager* for information about a real-time thread.

A real time thread is a subclass of *RTThread* or *RTJoinableThread*. The code that is actually executed by the thread is in the *Thread::startup*.

```
// A simple RT Program
// rt.h contains all the includes for the system
#include ``rt.h''
class MYThread : public RTJoinableThread {
public:
   // Notice that Thread's constructor must be called.
   // This happens because Thread is a virtual base class.
   // A virtual base class's constructor can be called by
   // multiple objects, with different parameters.
   // The compiler requires that the concrete class specify
   // parameters to resolve ambiguities.

   MYThread (RTParams p)
            :RTJoinableThread (p),
    Thread(Thread::joinable, (Thread::ThreadAttr) p){}
   // The code here is executed by the thread
   void startup() {
      stop(4000); // this tells the thread to stop for 4000usec
      while(1) {
        stop(1000); // this tells the thread to stop for 1000usec.
      };
   }
};


int main() {
   // At this point the RTManager has been initialized
   // and G_mother is operational.
   // First define an RTParams structure to pass into
   // MYThread's constructor
   // Note that the total time quantum must be equal or less
   // than one second
   RTParams p ( 0, // real time priority
        0, // time quantum in seconds
        500000, // time quantum in nanoseconds
                0 // LWP ID
               )

   MYThread thread(p); // this creates the RTThread object

   thread.make_runnable(); // this creates the real-time Solaris
                           // thread and starts it up
                           // using G_mother.

   exit(1); // this terminates the process
};
```

The class *MYThread* takes an *RTParams* as a parameter. The *RTParams* structure contains the time quantum and the priority. The time quantum is in seconds and nanoseconds. The total time quantum, nanoseconds plus seconds, must be less than or equal to one second. The priority must be between 0-57. If the time quantum is set incorrectly an exception is thrown inside of *G_rt_manager->ResetParams()*, and if the priority is not in the correct range an exception of type *RTError* inside of *RTThread::_InitThread* is thrown. The thread is still created but is in a suspended state. The thread parameters can be reset again by calling *G_rt_manager->ResetParams()* directly with the correct parameters. The *lwp_field_* of *RTParams* is set inside of *RTThread::_InitThread*.

Information about the real-time properties of a thread can be acquired through the singleton instance of *RTManager*, *G_rt_manager*. It is important that the user of the RTTP not try to create another instance of the *RTManager* class, since as part of *RTManager* initialization it creates the mother and dead-man switch.

6.3 Compiling a Program

Currently the RTTP only works for Solaris 2.4 and 2.5 because of the use of the Rogue Wave Tools++ library and because the Brown Thread Package is only implemented on top of the Solaris thread package. The Rogue Wave Tools++ will be replace by STL when SUN CC also compiles the STL.

To compile a program using the RTTP, you need the real-time library, librt.a, the shared C++ thread library, libc++thread.so, the dynamic linker library, libdl.so and the Rogue Wave Tools++ library, librwtool.a.

Any file that uses any component of the RTTP must include the rt.h.

To inhibit the dead-man switch, the RT_NDEAD_MAN flag must be defined.

A sample make line:

```
testrt: rt_test.C
       CC -g -mt -o testrt rt_test.C -I$INCDIR

       -R $LOC -L $LOC -ldl  -lrt -lrwtool  -lc++thread
```

7 Appendix B: Source Listing

```
#ifndef RTPARAMS_HEADER
#define RTPARAMS_HEADER

#include <sys/lwp.h>
#include <rw/tvslist.h>
#include <rw/tpslist.h>
#include <sys/rtpriocntl.h>
#include "c++thread.H"
#include <iostream.h>
#include <sys/types.h>
#include <FirstObj.H>

class RTParams : public Thread::ThreadAttr{
public:
        pri_t real_time_priority_;
        ulong time_quantum_;
        ulong nanoseconds_;
        lwpid_t lwp_;
        RTParams(RTParams& p)
                :real_time_priority_(p.real_time_priority_),
                 time_quantum_(p.time_quantum_),
                 nanoseconds_(p.nanoseconds_),
                 lwp_(p.lwp_),
                 Thread::ThreadAttr(Thread::ThreadAttr::BOUND |
                                    Thread::ThreadAttr::PROCESSOR,
                                    p.stack_size_,
                                    p.stack_buf_, p.pid_) {;}
        RTParams(pri_t real_time_priority,
                 ulong time_quantum,
                 ulong nanoseconds,
                 lwpid_t lwp,
                 long bound = BOUND | PROCESSOR,
                 processorid_t pid = 0,
                 size_t stack_size = 0,
                 char* stack_buf = 0)
                : Thread::ThreadAttr(bound,stack_size,stack_buf,pid),
                  real_time_priority_(real_time_priority),
                  time_quantum_(time_quantum),
                  lwp_(lwp), nanoseconds_(nanoseconds) {;}

};
#endif
```

```
#ifndef RTMANAGER_HEADER
#define RTMANAGER_HEADER

#include <sys/lwp.h>
#include <rw/tvslist.h>
#include <rw/tpslist.h>
#include <sys/rtpriocntl.h>
#include "c++thread.H"
#include <iostream.h>
#include <sys/types.h>
#include <FirstObj.H>
#include <RTParams.H>
#include <RTMother.H>

class RTError {};
int operator==(RTParams& left, RTParams& right);
class RTManager;

class DeadManSwitch : public RTThread
{
        RTManager* _manager;
public:
        DeadManSwitch (RTManager* manager = 0,
                        RTParams p = RTParams(58,0,100,0))
                :RTThread (p),
                 Thread(Thread::joinable, (Thread::ThreadAttr) p),
                 _manager(manager)
        { }
protected:
        void _InitThread(long) {
                Thread::_InitThread(flags);
        }
        void _atStartup();
        void startup();
};

class DeadManThread : public Thread {
        RTManager* _manager;
public:

        DeadManThread (RTManager* manager = 0)
                : _manager(manager){;}
        void startup();
};

class RTManager
```

```
{
        enum {MAX_DIGITS = 24};
        char* _path;
        pid_t _npid;
        lwpid_t _first;
        char * _pid;
        RWTValSlist<lwpid_t> rt_lwps_;
        RWTValSlist<RTParams> rt_params_;
        RWTPtrSlist<RTThread> rt_threads_;
        Mutex lock_;
        Mutex other_lock_;
        int _fileds;
        id_t* getID();
        Semaphore _deadman_sema;
        int _deadman;

        DeadManThread _dm;
        DeadManSwitch _ds;
        int _bit;
        void setBit() { _bit = 1;}
        int testBit() { int my_bit = _bit; _bit = 0; return my_bit;}

        void _AddThread(RTThread* rt_threads, RTParams& params);
        friend class DeadManThread;
        friend class DeadManSwitch;
        friend class RTMother;
        void AddThread(RTThread* rt_threads, RTParams& params);

public:
#ifndef RT_NDEAD_MAN
        RTManager(char* path = "/usr/bin/rtpriocntl",
                  int deadman = 1);
#else
        RTManager(char* path = "/usr/bin/rtpriocntl",
                  int deadman = 0);
#endif
        ~RTManager();
        void Query(RTParams& params);
        void RemoveThread(RTParams& params);
        void ResetParams(RTParams& params);
};

static FirstObj <RTManager> G_rt_system;

#endif
```

```
#include "RTManager.H"
#include <string.h>
#include <sys/lwp.h>
#include <iostream.h>
#include <string.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <sys/wait.h>
#include <sys/signal.h>
#include <sys/fault.h>
#include <sys/procfs.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <Debug.H>
#include <assert.h>

char * RTFile = "/usr/bin/rtpriocntl";
const char * RTset = "-s";
const char * RTclass = "-c";
const char * RT = "RT";
const char * RTtqtm = "-t";
const char * RTres = "-r";
const char * RTpri = "-p";
const char * RTid = "-i";
const char * RTpid = "pid";
extern RTMother* G_mother;
int operator==(RTParams& left, RTParams& right){
        return left.lwp_ == right.lwp_;
}
extern int errno;

RTManager::RTManager(char* path, int deadman)
        :_deadman_sema(0),
         _dm(this),_ds(this)

{
                _bit = 1;
                _deadman = deadman;
                _npid = getpid();
                _first = _lwp_self();
```

```
                char* env_path = getenv("PRIOLOC");
                if (env_path)
                        path = env_path;
                _path = new char[strlen(path)+1];
                strcpy(_path,path);

                _pid = new char[24];
                memset(_pid,0,24);
                sprintf(_pid,"%d",_npid);



                char c[256];
                sprintf(c,"/proc/%d",getpid());
                if((_fileds = open(c, O_RDONLY)) == -1){
                        perror("could not open /proc file");
                        exit(1);
                }



                if(deadman) {
                        _dm.make_runnable();
                        _ds.make_runnable();
                } else {
                        _deadman_sema.post();
                }

                G_mother = new RTMother;
                G_mother->make_runnable();


}


RTManager::~RTManager()
{
}

id_t* RTManager::getID()
{

        prstatus pstatus;

                int ret_val = ioctl (_fileds,PIOCSTATUS,&pstatus);
```

```
                if (ret_val == -1) {
                        perror("failure of ioctl");
                        exit(1);
                }


        id_t* ids = new id_t[pstatus.pr_nlwp];

                int ret_val = ioctl (_fileds,PIOCLWPIDS,ids);
                if (ret_val == -1) {
                        perror("failure of ioctl");
                        exit(1);
                }

        return ids;

}

void RTManager::AddThread(RTThread* rt_thread, RTParams& params)
{

    lock_.lock();
            _deadman_sema.wait();
            _AddThread(rt_thread,params);
            _deadman_sema.post();
    lock_.unlock();

}

void RTManager::_AddThread(RTThread* rt_thread, RTParams& params)
{
        // first we have to make sure we are the real pid;
        // add this to the list of real time lwps

        if(_npid ≠ getpid()) {
          return;
        }

                rt_lwps_.insert(params.lwp_);
                rt_params_.insert(params);

                char stqtm[RTManager::MAX_DIGITS];
                char res[RTManager::MAX_DIGITS];
                char pri[RTManager::MAX_DIGITS];

                memset(stqtm,0,RTManager::MAX_DIGITS);
```

```
memset(res,0,RTManager::MAX_DIGITS);
memset(pri,0,RTManager::MAX_DIGITS);
int tqtm = 100;
sprintf(stqtm,"%d",tqtm);
int resolution = 1000;
sprintf(res,"%d",resolution);
sprintf(pri,"%d",17);

pid_t child = fork1();


        if (child == 0) {

                // we are in the child
        execl(_path, _path, RTset, RTclass, RT, RTpri, pri,
                RTtqtm, stqtm, RTres, res,RTid, RTpid,
                _pid, 0);
        assert(1==0);
        exit(1);

        }


int stat_loc;
if(wait(&stat_loc) ≠ child)
        throw (RTError());

if(WEXITSTATUS(stat_loc) ≠ 0)
        throw (RTError());


id_t* array = getID();

for(;*array ≠ 0; array++){
        lwpid_t value;
        if(!rt_lwps_.find(*array,value)){
                try {
                        RTParams p(0,0,0,*array);
                        RemoveThread(p);
                } catch(...){
                      fprintf(stderr, "Bad news %d \n ",
                              *array);
                }
        }
}
```

```
                for(int i = 0; i < rt_params_.entries(); i++)
                        ResetParams(rt_params_[i]);
                delete [] array;

                rt_threads_.insert(rt_thread);

}

// this puts the lwp off the RT band
void RTManager::RemoveThread (RTParams& p)
{

        lwpid_t lwp = p.lwp_;

        pcparms_t pcparams;
        pcinfo_t pcinfo;


        short maxupri;
        strcpy (pcinfo.pc_clname,"TS");
        if (priocntl (0,0,PC_GETCID,(caddr_t) & pcinfo)
            ==-1) {
                perror("priocntl removal ");
                throw (RTError());
        }

        maxupri = ((tsinfo_t *)pcinfo.pc_clinfo)->ts_maxupri;


        pcparams.pc_cid = pcinfo.pc_cid;
        ((tsparms_t *)pcparams.pc_clparms)->ts_uprilim = TS_NOCHANGE;
        ((tsparms_t *)pcparams.pc_clparms)->ts_upri = TS_NOCHANGE;

        if (priocntl(P_LWPID, lwp ,PC_SETPARMS,
                        (caddr_t) &pcparams)==-1) {
                perror("priocntl removal2 ");
                throw (RTError());
        }

}

void RTManager::ResetParams(RTParams& params)
{
```

```
        other_lock_.lock();


                short rt_pri = params.real_time_priority_;
                ulong rt_tqsecs = params.time_quantum_;

                pcparms_t pcparams;
                pcinfo_t pcinfo;

                strcpy(pcinfo.pc_clname,"RT");
                if (priocntl(0,0,PC_GETCID,(caddr_t) & pcinfo)==-1) {

                        perror("prionctl reset " );
                        other_lock_.unlock();
                        throw(RTError());

                }

                pcparams.pc_cid = pcinfo.pc_cid;
                ((rtparms_t *)pcparams.pc_clparms)->rt_pri = rt_pri;
                ((rtparms_t *)pcparams.pc_clparms)->rt_tqsecs = rt_tqsecs;
                ((rtparms_t *)pcparams.pc_clparms)->rt_tqnsecs =
                        params.nanoseconds_;

                int success = 0;
                int res = 0;
                int counter = 0;
                if ((res = priocntl(P_LWPID,params.lwp_,
                                        PC_SETPARMS,
                                        (caddr_t) &pcparams)) == -1) {
                        int err = errno;
                        perror("and the theme is ");
                        other_lock_.unlock();
                        throw RTError();
                }

        other_lock_.unlock();

}

void RTManager::Query(RTParams& params)
{
        lock_.lock();
        pcparms_t pcparams;

                short rt_pri = params.real_time_priority_;
```

```
                ulong rt_tqsecs = params.time_quantum_;


                memset(&pcparams,0,sizeof(pcparms_t));
                pcparams.pc_cid = PC_CLNULL;

                if (priocntl(P_LWPID,params.lwp_, PC_GETPARMS,
                                (char*) &pcparams) == -1) {

                        perror("Error in Queury");
                        other_lock_.unlock();
                        throw RTError();

                }

        lock_.unlock();


        params.real_time_priority_ =
                ((rtparms_t*) pcparms.pc_clparms)->rt_pri;
        params.time_quantum_ =
                ((rtparms_t *)pcparms.pc_clparms)->rt_tqsecs;
        params.nanoseconds_ =
                ((rtparms_t *)pcparms.pc_clparms)->rt_tqnsecs;
}

void DeadManThread::startup()
{
        while (1) {

                if(!_manager)
                        G_rt_system->setBit();
                else
                        _manager->setBit();

                struct timeval tv;
                tv.tv_usec = 50000;
                tv.tv_sec = 0;
                select(0,0,0,0,&tv);

        }

}

void DeadManSwitch::_atStartup()
{}
```

```
void DeadManSwitch::startup()
{
        params_.lwp_ = _lwp_self();

        if (!_manager) {
                G_rt_system->_AddThread(this,params_);
                G_rt_system->_deadman_sema.post();
        } else {
                _manager->_AddThread(this,params_);
                _manager->_deadman_sema.post();
        }

        while (1) {
                struct timeval tv;
                tv.tv_usec = 0;
                tv.tv_sec = 1;
                select(0,0,0,0,&tv);

                if (!_manager->testBit())
                        ::exit(1);
        }

}
```

```
#ifndef RTMOTHER_HEADER
#define RTMOTHER_HEADER
#include <c++thread.H>

#include <FirstObj.H>
#include <RTParams.H>
#include <RTThread.H>

class RTMother : public RTThread
{
        RTMother& operator=(const RTMother& mother);
        RTMother(const RTMother& mother);

public:

        RTMother (RTParams p = RTParams(50, 0, 3 * 100000000, 0));
        ~RTMother () {};

public:
        void GiveBirth (long flags,
                        RTThread* thread);

protected:
        virtual void startup ();
        virtual void _atStartup ();
        virtual void _InitThread(long) {
                Thread::_InitThread(flags);
        }

protected:
        long _flags;
        Mutex _thread_lock;
        Semaphore _give_birth;
        Semaphore _gave_birth;
        RTThread* _thread;
};

#endif
```

```
#include "RTMother.H"
#include <RTManager.H>
RTMother* G_mother;

RTMother::RTMother(RTParams p)
        : RTThread (p),
          Thread(Thread::joinable, (Thread::ThreadAttr) p),
          _give_birth(0),
          _gave_birth(0)

{;}

voiid RTMother::GiveBirth(long flags,
                          RTThread* thread)
{

        _thread_lock.lock();
        fprintf(stderr,"IN give birth \n");
        _flags = flags;
        _thread = thread;
        _give_birth.post();
        _gave_birth.wait();
        _thread_lock.unlock();

}

void RTMother::_atStartup()
{

        params_.lwp_ = _lwp_self();
        G_rt_system->AddThread(this, params_);

}

void RTMother::startup()
{

        while (1) {
                _give_birth.wait();
                fprintf(stderr,"IN startup\n");
                _thread->InitThread(_flags);
                _gave_birth.post();
        }

}
```

```
#ifndef RTTHREAD_HEADER
#define RTTHREAD_HEADER
#include <c++thread.H>
#include <RTParams.H>

class RTThread : virtual public Thread
{
protected:
        RTParams params_;

protected:
        virtual void atExit();
        virtual void _atStartup();
        virtual void _InitThread(long flags);

protected:
        virtual void stop(time_t duration);

public:
        RTThread (pri_t priority, ulong tqtm = 100);
        RTThread (RTParams& p);
        virtual ~RTThread();

public:
        virtual void InitThread(long iflags) {
                Thread::_InitThread(iflags);
        }
};

class RTJoinableThread : public RTThread, virtual public JoinableThread
{
public:
        RTJoinableThread (pri_t priority, ulong tqtm = 100);
        RTJoinableThread (RTParams& p);
        ~RTJoinableThread(){};

public:
        virtual void InitThread(long iflags) {
                Thread::_InitThread(iflags);
        }

};
#endif
```

```
#include <c++thread.H>
#include <sys/lwp.h>
#include <assert.h>
#include <RTThread.H>
#include <RTParams.H>
#include "RTManager.H"
#include <RTMother.H>
extern RTMother* G_mother;
RTJoinableThread::RTJoinableThread(pri_t priority, ulong tqtm)
        :JoinableThread(
Thread::ThreadAttr(Thread::ThreadAttr::BOUND)),
         Thread(Thread::joinable,
                Thread::ThreadAttr(Thread::ThreadAttr::BOUND)),
         RTThread(priority,tqtm)
{}

RTJoinableThread::RTJoinableThread (RTParams& p)
        :JoinableThread ((Thread::ThreadAttr) p),
         Thread (Thread::joinable, (Thread::ThreadAttr) p),
         RTThread(p)
{}

RTThread::RTThread(pri_t priority, ulong tqtm)
        :Thread(Thread::joinable,
                Thread::ThreadAttr(Thread::ThreadAttr::BOUND)),
         params_(0,0,0,0)
{
        params_.bound_ = attr_.bound_;
        params_.real_time_priority_ = (priority);
        params_.time_quantum_= (tqtm) / 1000;
        params_.nanoseconds_ = (tqtm) * 1000000;
}

RTThread::RTThread (RTParams& p)
        :Thread (Thread::joinable, (Thread::ThreadAttr) p),
         params_ (p)
{}

void RTThread::_InitThread (long flags)
{
        G_mother->GiveBirth(flags, this);
        if(params_.real_time_priority_ > 57)
                throw RTError;

}
```

```
void RTThread::_atStartup()
{

        params_.lwp_ = _lwp_self();
        G_rt_system->ResetParams(params_);

}

void RTThread::atExit()
{

        G_rt_system->RemoveThread(params_);

}

RTThread::~RTThread()
{}

void RTThread::stop(time_t duration)
{
        struct timeval tv;
        tv.tv_usec = duration;
        tv.tv_sec = 0;
        select(0,0,0,0,&tv);
}
```

References

[Gad] Frank Gadegast. *MPEG FAQ*. PHADE SOFTWARE.

[HJT+93] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: A case study. *Operating Systems Review*, pages 94–105, December 1993.

[Jr] Thomas W. Doeppner Jr. *The Brown C++ Threads Package*. Brown University.

[Suna] SunSoft. *man Pages(1): User Commands*.

[Sunb] SunSoft. *man Pages(2): User Commands*.

[Sunc] SunSoft. *Multithreaded Programming Guide*.

[Sund] SunSoft. *System Services Guide*.