TRACING AND REPLAY OF REAL-TIME CONCURRENT PROGRAMS IN VxWorks

by

Daxa Keshavji Patel

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington and
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON
December 2006

ACKNOWLEDGEMENTS

I am grateful to Dr.Lei for helping me tremendously in my work and for giving me the opportunity to work under him. I am thankful to Dr.Kung and Dr.Fegaras for being the committee members for my defense. I would also like to thank Abhinav Agrawal and all the members of the Software Engineering Research Group led by Dr.Lei who helped me during my work.

I would like to acknowledge all my friends who encouraged and motivated me throughout my work. Last but not the least, I would like to mention my family who was my source of inspiration and who gave me constant moral support throughout my studies.

November 27, 2006

ABSTRACT

TRACING AND REPLAY OF REAL-TIME CONCURRENT PROGRAMS IN VxWorks

Publication No. _____

Daxa Keshavji Patel, M.S.

The University of Texas at Arlington, 2006

Supervising Professor: Dr.Jeff Yu Lei

Real-time concurrent programs are difficult to analyze, debug and test because of the existence of race conditions. In particular, cyclic debugging requires the ability to reproduce a failed execution. That is, when a failure is observed during a test run, it is often necessary to reproduce the test run as an effort to locate the bug that has caused the failure. However because of the variations in thread scheduling and signal latency, a real-time concurrent execution may not be reproduced simply by re-executing the program under test. In this thesis, we describe a language-based framework for tracing and replay of real-time concurrent programs in Vxworks. The framework consists of wrappers for Vxworks synchronization constructs like semaphores, message queues and threads. This framework supports two modes of execution, namely, trace and replay. In the trace mode, important synchronization events, with necessary debugging information, are recorded into a trace file. In the replay mode, a trace of synchronization events is read and used to control the behavior of threads so that these events are exercised in the same order as they were recorded. The ability of tracing and replay real-time concurrent executions

facilitates the dynamic analysis and debugging of these executions. Unlike most existing frameworks, which are implemented in a platform-specific manner, our framework inserts additional runtime control at the programming language level and can be easily ported to other platforms.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

The correctness of real-time software depends not only on the logical results of its computations but also on whether those computations satisfy certain timing requirements. Real-time software developers need to make sure that their software does right computations at the right times. Many real-time systems are concurrent by nature and multiple threads may introduce non-determinism in the system which may lead to different output every time the program is run with the same input parameters for debugging. This non-determinism is introduced because the threads in the system may change their interleavings every time the program is executed. These interleavings can change the order of events or the time of occurrences of events in the program during each run.

For example, consider a concurrent program with 2 threads t1 and t2; this is an example which illustrates the concept of non-determinism using a simple application which initializes and/or increments the value of a variable. Access to the shared variable is mutually exclusive using semaphore operations $semTake()$ and $semGive()$. The thread requests for a permit to access the shared variable using $semTake()$ and it releases the permit using $semGive()$ so that the other thread can access the shared variable.

If the threads request for the shared variable access in the order shown in figure 1.1, the value of the variable x read by thread t1 is 1 whereas the value read by thread t2 is 2. Now, consider the other possible interleaving of threads as shown in figure 1.2.If threads t1 and t2 execute the instructions in this order, the value of the shared variable x read by thread t1 is 2 whereas the value read by thread t2 is 1.

```
Assume the shared variable x is initialized to 0;
   Thread t1:                          Thread t2:
   1  semTake();
   2                                   semTake();
   3  x++;//x=1
   4  semGive();
   5                                   x++;//x=2;
   6                                   semGive();
```

Figure 1.1 One possible interleaving of threads in the sample program

```
Assume the shared variable x is initialized to 0;
   x=0;
    Thread t1:                         Thread t2:
1
2                                      semTake();
3  semTake();
4                                      x++;//x=1;
5                                      semGive();
6  x++;/x=2;
7  semGive();
```

Figure 1.2 The other possible interleaving of threads in the sample program

Now consider another example with three threads t1, t2 and t3. These threads execute events e1, e2 and e3 respectively and the times at which these events are executed are also of significance. These three threads are scheduled to execute events as shown in the figure 1.3.

It is possible that while execution, switching between these threads may not occur in the same way every time. Because of thread priority or other factors that may affect threads blocking and unblocking, they may exercise different schedules every time the program is run. As shown in the Figure 1.4 above, because of an early switching between the thread t2 and the thread t3, the order and the time at which the events should execute are also different than in the reference schedule shown in the figure 1.3.

So, the common approach of debugging the program by replaying it multiple times and analyzing it to look for errors does not work for real-time concurrent programs. Also, interactive debugging of real-time programs without deterministic replay is not

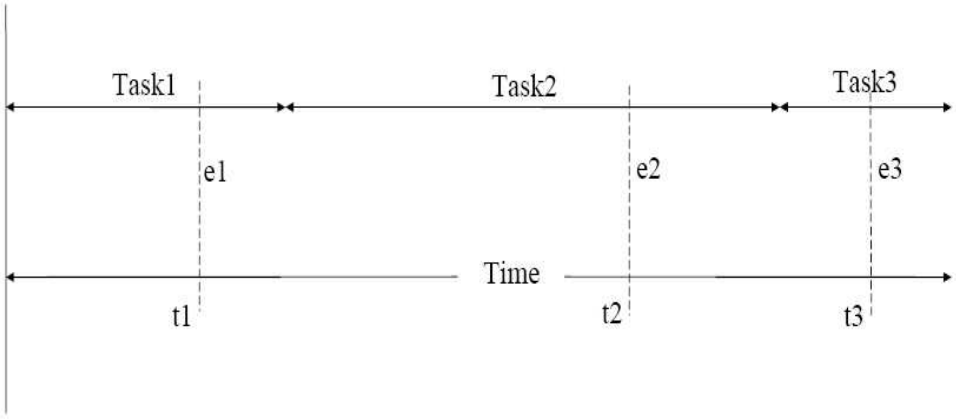

Figure 1.3 Reference schedule of three threads (tasks)

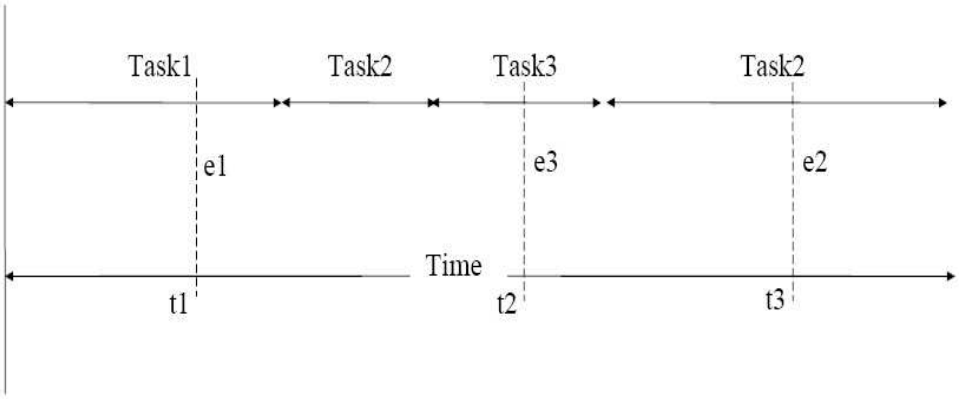

Figure 1.4 Another interleaving of threads due to an early task switch

sufficient because debugging commands can destroy the timing-dependent nature of real-time systems.

Our work focuses on tracing and replay of real-time concurrent programs in VxWorks. VxWorks is the one of the most widely used operating system for real-time software development. It provides various APIs for developing real-time software. The development environment used is Wind River, a general purpose platform for VxWorks.

Multiple threads in a program written in VxWorks may communicate with each other using various synchronization constructs like semaphores, message queues, inter-

rupts, events, task creation and communication facilities. Our approach is to develop API wrappers for these synchronization constructs. Using these API wrappers, we can control the total ordering based sequence and timing of threads execution to replay the exact scenario.

To carry out deterministic execution of programs using our approach, we need to determine:

- The minimum and sufficient information to carry out replay and further debugging,
- How to efficiently collect this information and
- How to use this collected information to replay the same erroneous execution of the program. That is how to use this information to control the progress or the schedule of threads to carry out the same erroneous behavior again.

This work contributes towards providing a framework for tracing and replay of multi-threaded VxWorks programs. This approach could also be used for incremental debugging. The incremental debugging approach means that a set of events are tested first, then the other set of events, and so on until the bug is found. The process of developing a set of test problems can often take half of the debugging effort.

1.2 Structure of Thesis

The rest of the thesis is organized as follows.

Following is the related work done in this area. Chapter 3 gives overview of the VxWorks operating system and its main API features like tasks, semaphores, message queues and I/O operations. Chapter 4 represents the approach used followed by a sample experiment. To conclude with, following is conclusion, future work and references used for this work.

CHAPTER 2

RELATED WORK

Repeated re-execution is the widely accepted technique for debugging deterministic sequential applications. Repeated re-execution for debugging however fails to work for non-deterministic multi-threaded applications because the bug that manifested itself during one execution instance might not manifest itself again in another execution instance. To use repeated re-execution technique to find bug in nondeterministic systems, there have been several approaches practiced as mentioned below:

Past approaches to replay shared-memory applications have focused on recording how processes interact. The most straight-forward way of recording an execution is to save the value of every shared read operation [1]. R. Netzer [2] in his paper gives an approach that is designed for the monitoring of every shared-memory access.

The obvious draw-back of this approach is that if sharing is common, trace log size will rapidly explode. In order to minimize the amount of data saved, the technique should save the order of shared accesses, rather than the content.

Event ordering is the approach used by many replay tools for concurrent programs. In this approach, the events that cause interaction between threads, which are access to shared data and synchronization are controlled.

In [3], a debugging environment is developed which consists of two run-time systems (event monitoring system, replay system) and a concurrent program debugger. The run-time systems record events causing non-deterministic behavior of concurrent programs and replays the program following the execution order of the original execution using the recorded events. Concurrent program debugger replays the error-occurred execution

supported by the replay system and debugs the errors in source level. In this paper, they treat the concurrent programs which only use explicit synchronization primitives, synchronous/asynchronous message passing primitives, critical section, and shared variables. Each process is independently executed and can communicate with other processes by the reference of shared variables and the messages send/receive.

[4] suggests a system-independent language-level approach to deterministic execution replay for concurrent Ada programs. Synchronization events are used to replay the monitored application. Their approach records the synchronization (SYN-sequence) S of a concurrent Ada programs P(X) and uses this sequence to transform P to a concurrent Ada program $P(X, S)$, which produces the same output.

The approach shown in [5] is based on executing the intermediate form of the concurrent program. They use a special function that provides information about thread interactions by identifying the concurrent reaching definitions at a variable use. For variable read, writes from other threads that may be returned, presented as terms in the special function before the use. The function with multiple terms highlight places in the code where the thread schedule may alter the outcome, as the value for a variable use depends on the order in which threads are run. While replaying, the same function is used to control the access to variable use by the threads in the specified order.

However, approaches in [3], [4] and [5] can not be used directly for real-time systems. They lack in the respect that they can only replay concurrent program execution events like rendezvous, do not support task switching as synchronization events, and they do not consider issues such as real-time clock values or dynamic task creation.

There are some approaches that also replay task-switches and hence provide significant control over the schedule which is one of the important features of real-time system.

To replay and debug multitasking real-time systems using Time Machine based approach in [6], they also monitor the system control-flow in addition to the data-flow. Essentially, the control-flow corresponds to a list of task switches, that is, all transfers of control from one task to another task and back. To make the approach general they store the values of stack pointer, register bank checksums, and/or parts of the checksums of the user-stack. The approach is to store the values of the data variables and assign these values to the corresponding variables while replay. However it has the disadvantage discussed in the beginning of this chapter. That is, instead of storing the order in which the data is accessed, it stores the information about data itself which explodes the trace log size. Also, manual instrumentation has to be inserted into the source code to monitor each data access in order to capture values of the individual data variables. This approach also does not provide significant control over the schedule as it controls the order of task switches but it does not control the order in which these tasks access the data.

Our approach for replay combines the approaches of synchronization events order replay and task switch replay. It combines the advantages of less memory storage for storing events order instead of information about each shared variable and schedule control with acceptable accuracy.

CHAPTER 3

VxWorks

3.1 VxWorks Overview

Real-time operating systems are developed to perform a specialized set of tasks, and have a strict set of requirements for their operating systems [7]. VxWorks is the one of the most widely implemented real-time operating systems. VxWorks can make scheduling guarantees that normal operating systems cannot. VxWorks and Tornado, a set of tools used for application development, are an integral part of this thesis. Real-time operation is a very important part of this thesis, and this is why VxWorks, one of the popular real time operating systems (RTOS), and its associated Integrated Development Environment (IDE), Tornado, have been used.

The reason why a real-time operating system is used is that the traditional operating systems like Windows and UNIX are ill suited to handle any real-time applications.

On the other hand, operating systems like VxWorks, built especially for handling real-time processes, are poor performers when it comes to non real-time application development. VxWorks can operate in tandem with either UNIX or Windows so that each type of operating system (RTOS and non-RTOS) can do what it does the best. This means that VxWorks would handle the time-critical aspects while the host operating system would handle the program development and non real-time aspects of the application. VxWorks is very flexible. It lets a user customize it to include the features that the application requires. For example, VxWorks allows the user to configure it such that the features like networking can be added to speed up the development cycle, and the same can be excluded in the post-production phase. Thus, we see that a real-time operating

system has significant advantages over the traditional operating systems when it comes to working with sensitive timing requirements. The basic requirements and capabilities of VxWorks are summarized below:

- Real-time operating systems should always respond to an event in a guaranteed amount of time, which is in the order of microseconds or nanoseconds. This is the essence of real time.
- Real-time operating systems are capable of working with a minimal set of resources, like memory.
- A real-time operating system is rugged and will be resilient to situations that would otherwise cause a non real-time operating system to crash.

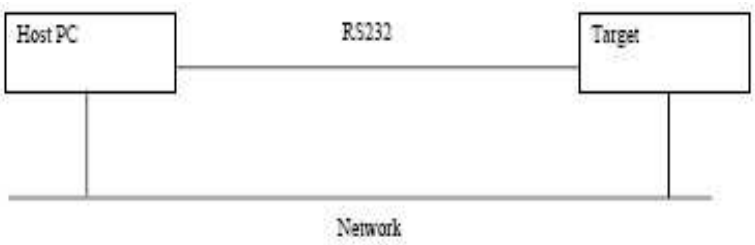

Figure 3.1 Real-time Operating Systems

Figure 3.1 shows how a real-time operating system fits into the scheme of things, with the development tools running on a host computer, and VxWorks running on the development board.

The real-time operating system will run on the hardware target, while the development tools run on the host computer. The network functions as the interconnection between the host and the target. It provides facilities like file transfers between the host and the target, and has the ability to boot the board over the network. The development

tools communicate with the real-time operating system and gather information that can be analyzed.

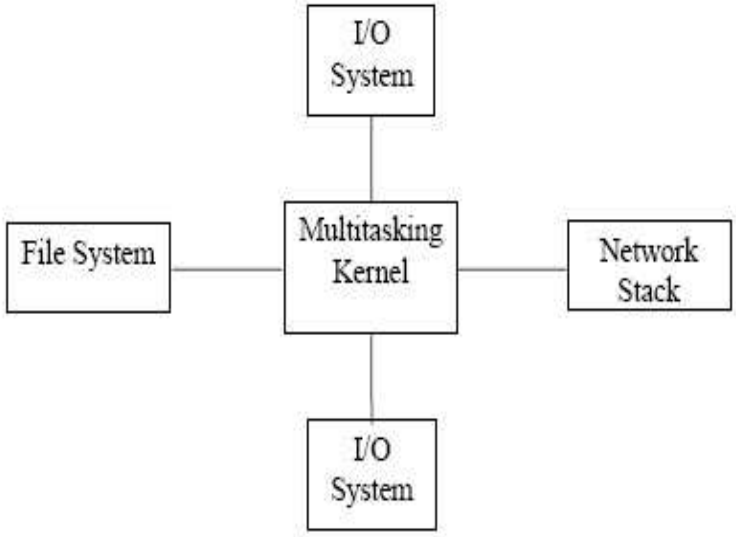

Figure 3.2 Real-time Operating Systems

The functional structure of real-time operating systems comprises of a multitasking kernel, which performs functions like real-time scheduling, intertask communication and mutual exclusion. This kernel is linked to the file systems, device drivers, I/O systems, and the network stack. More information about the various facilities offered by VxWorks is described below. The subsequent sections will deal with how VxWorks is used in real-time development.

VxWorks RTOS Facilities

Real-time systems, as mentioned above, are designed to work in conjunction with standard operating systems. The following is then brief overview of the specific facilities and tools provided by VxWorks [Wv99]. Facilities:

- High Performance Real-time kernel facilities: VxWorks' kernel, called 'Wind' is a multi-tasking kernel, and its key features are real-time scheduling, mutual exclusion (semaphores), and intertask communication. These features are what distinguish the real-time operating system from the non real-time operating system. The kernel is very fast and engages in preemptive priority based scheduling of its tasks. This kernel belongs to what we would call microkernel architecture, for it is small and highly configurable.
- Shared Memory Objects: In the case of a multi-processing environment, VxWorks has features that enable it to share resources like memory and semaphores between different processors.
- Fast file systems: VxWorks employs custom tailored file systems that are designed for real-time operation. The file system formats used are compatible with DOS, RT-11, SCSI, CD-ROM, and "raw-disk" file system. Overall, these fast file systems are critical to the operations of a RTOS, where timing is of the essence.
- C++ support: VxWorks ships with a variant of GNU's GCC C/C++ compiler.
- There are other compilers available, like the Diab compiler, which is WindRiver's own compiler.
- Input/Output System: VxWorks comprises a speedy, stable and flexible I/O system. It includes several drivers that encapsulate key input/output mechanisms, like keyboard driver, RAM Disk driver, display driver, etc.
- Target Resident Tools: A complete set of tools used for development are present on the host machine, thus conserving target memory and resources, but there is also a target resident shell, a symbol table, and a module loader/unloader that can be configured into the target OS as required.
- Evaluation Tools and Utility libraries: Performance evaluation tools that include utilities to display processor utilization percentages by a particular task, and an

execution timer are present. VxWorks includes a comprehensive set of utilities, like timers, interrupt handling, memory allocation, and ANSI C libraries.

- Network Facilities: VxWorks network facilities are compatible with standard internet protocols. Thus, setting up communications between the host and the target using regular equipments like router is a very simple process.
- Board Support Packages: A board support package is the board specific part of VxWorks. It contains initialization code for the hardware as well as device drivers for included components on the board like serial, parallel and ethernet ports, SCSI and IDE controllers, etc. VxWorks supports a large number of boards, and thus provides an easy development process.
- Virtual Memory: VxWorks includes virtual memory management for boards that require this feature.
- APIs: The following are the API libraries provided with VxWorks.
- Synchronization - Message Queues and Semaphores
- Task - Routines to create, resume, suspend, delay, lock and unlock tasks
- Kernel Space - To access hardware and processor level facilities
- Timer facilities - Watchdog and kernel timer in ticks
- I/O facilities - File and Console I/O

VxWorks Projects

The projects in VxWorks are used as logical containers and as building blocks that can be linked together to create a system [7].

There are various types of pre-configured templates for different project types which allow us to create or import projects using simple GUI.

Preconfigured Project Types:

- Workbench Sample Projects: There is set of some sample projects which provide a good walk through about the workbench.
- VxWorks Image Projects: This type is used to configure and build a kernel image for booting the target.
- VxWorks Board Support Packages: These types of projects are used to boot and load the target with the VxWorks Kernel. This process is called the boot loader. Boot loaders are not required for standalone VxWorks systems that are stored in ROM.
- VxWorks Downloadable Kernel Module Projects: These are specifically used for managing and building the projects that will exist in the kernel space. These modules can be build separately, as well as run and debug separately on the target running at VxWorks. Loading, reloading, unloading all can be achieved at the runtime. This development mode is the traditional VxWorks method for all the tasks spawned, and they run in an unprotected environment, and have full access to the underlying hardware. A downloaded Kernel Module that is linked into the kernel is a bootable application and starts as soon as the target is booted.

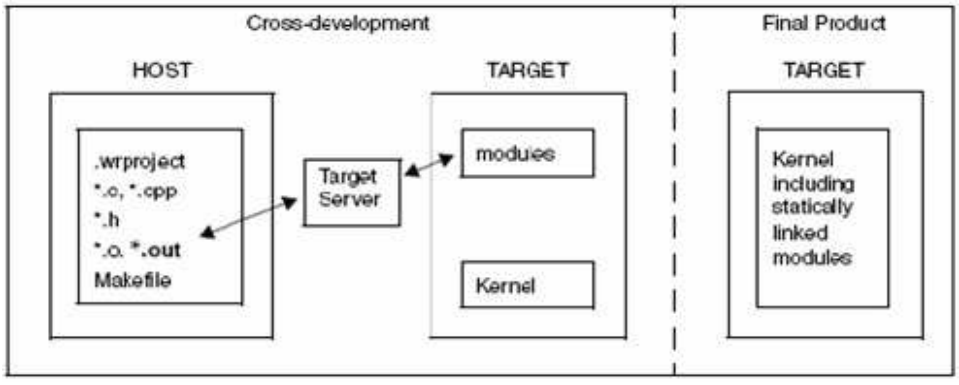

Figure 3.3 Downloadable Kernel Modules Structure

- Real-time Process Projects: These are specifically used for managing and building the projects that will exist outside the kernel space. These modules can be built separately, as well as run and debug separately on the target running at VxWorks. Loading, reloading, unloading all can be achieved at the runtime.

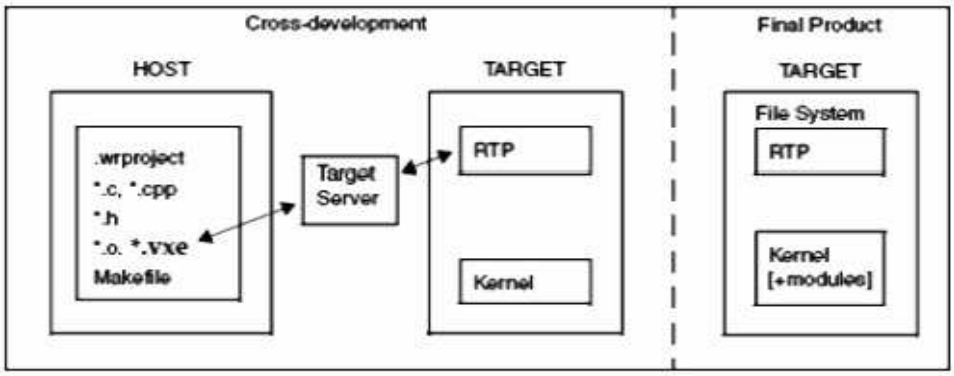

Figure 3.4 Real-time Processes Project

- VxWorks Shared Library Projects: Shared Library projects can be used for creating sub-projects that are statically linked into another project types at build time. At the same time VxWorks Shared Library projects can also be used for libraries that dynamically link to VxWorks Real-time Process project, on a target side file system.

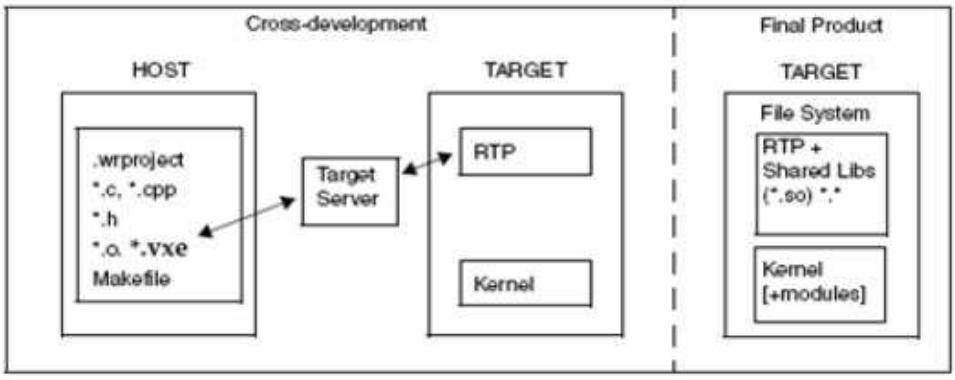

Figure 3.5 File System Projects

- VxWorks File System Projects: These can be used as a subproject of any other project type that requires target-side file system functionality. This type is specially designed for bundling applications and other files, of any type, with a VxWorks system image. The advantage of using such file type is that no storage media is required for the VxWorks boot image.

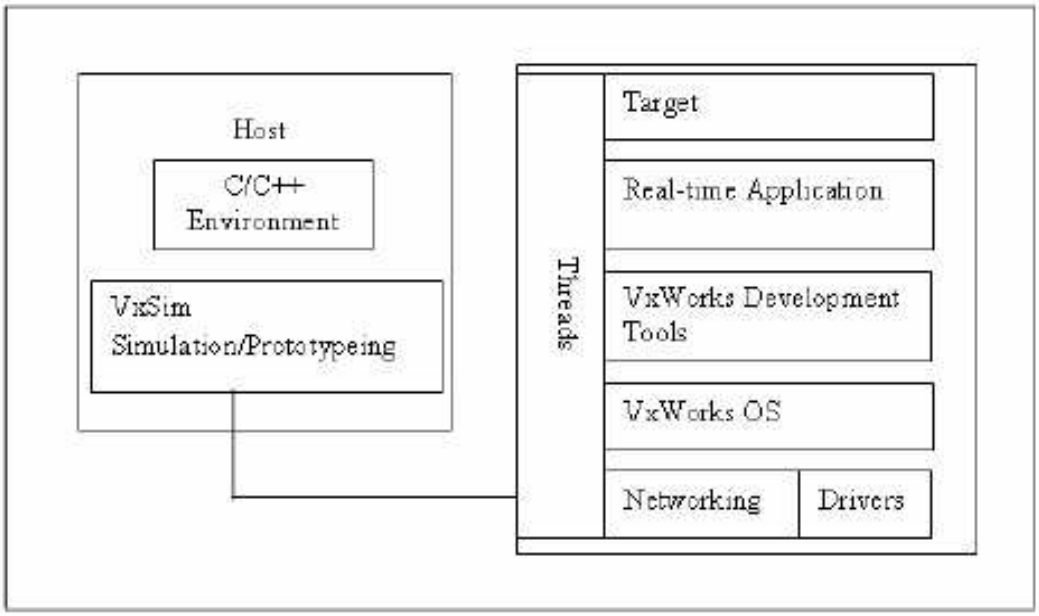

Figure 3.6 VxSim Architecture [WindRiver]

3.2 VxWorks simulator (VxSim)

There is a host based simulator that we can use to test our software when we do not have, or do not want to use a "real" target system. The simulator still appears to the tools to be a target, and connects to the IDE via a target server, but it runs as a separate process on the host.

It is a part of Wind River suite. VxSim provides full VxWorks simulation on a UNIX workstation. It enables application development to begin before hardware becomes available, and allows a large portion of software testing to occur early in the development cycle - prior to integration testing. VxSim accurately reproduces many of the sophisticated features of VxWorks; including the DOS file system, full UNIX-style networking (TCP/IP, rlogin, etc.) and support for up to 16 targets sharing a common backplane - plus simulation of the VxWorks Backplane Driver for inter-processor communication.

3.3 VxWorks API Features

3.3.1 Tasks

In VxWorks, the unit of execution is the task, corresponding to a Unix/Linux process or Java threads. Tasks may be spawned that is created; deleted, resumed, suspended, and preempted that is interrupted by other tasks, or may be delayed by the task itself. A task has its own set of context registers including stack. The term thread, while not unknown in VxWorks jargon, does not exist in a formal sense as in other operating systems. A thread, when the term is used, may be thought of as a sub-sequence of connected program steps inside a task, such as the steps the VxWorks kernel performs in spawning a task, or the sequence of instructions in an else-clause following an if-statement. Tasks can communicate with each other in a manner similar to Inter-process communications in Unix and Linux.

Tasks are in one of four states, as shown in figure 3.7. A newly spawned task enters the state diagram through the suspended state. Tasks may be scheduled for execution by assigning them priorities, ranging from 0 (highest priority) to 255. Once started, that is, having entered the ready state in Figure 3.7, a task may execute to completion, or it may be assigned a fixed time slice in round-robin scheduling. A task blocks that is

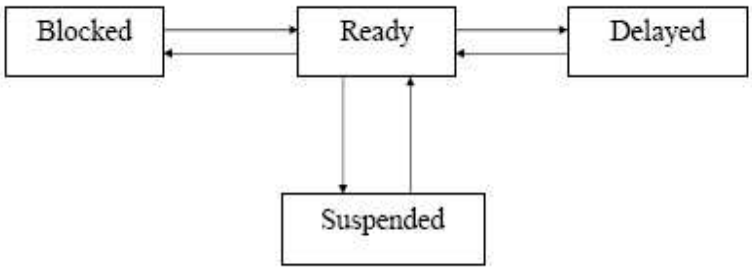

Figure 3.7 Task State Diagram

it enters the pended state while another task takes control. A task may be preempted because it has consumed its time slice, or because another task with higher priority exists. Task priorities may be changed during execution. A task may avoid being preempted by locking the scheduler while it has control. A task may also enter the delayed state, for example while waiting a fixed time between reading samples within a task before processing them as a group, during which time another task may take control. The delay is controlled by an external timer which runs independently of processing (combined with a tick counter maintained by the kernel) that awakens the delayed task and avoids having the task tie up resources with an index counter which would prevent another task from executing.

The suspended state is used to halt a task for debugging without loss of its context registers. Several system tasks exist concurrently with user defined tasks. These are the root task, named tUsrRoot; the logging task tLogTask; exception task tExcTask; and the network task tNetTask.

3.3.2 Semaphores

Semaphores can be categorized as ordinary binary semaphores and a special class of binary semaphores called mutual exclusion semaphores [7]. Binary semaphores are used for task synchronization. As implemented in VxWorks, a binary semaphore has

two values: full and empty. When full, it is available for a task. When empty, it is unavailable. A pending task proceeds by taking an available semaphore, which makes the semaphore empty or unavailable. When the semaphore is no longer needed because the task is about to return to the pending state, it gives the semaphore which makes it full or available for another task. A mutual exclusion semaphore also called a mutex allows one task to have exclusive use of a resource while it is needed.

The difference between an ordinary binary semaphore and a mutex semaphore is in the way the semaphore is initialized. For an ordinary binary semaphore, a task attempting to synchronize to an external event creates an empty semaphore. When it reaches the point to be synchronized, it attempts to take a semaphore. If unavailable, it waits at this point. A second task which controls the synchronization event gives the semaphore when it is no longer needed. The first task receives notification that the semaphore is available and proceeds. For a mutex semaphore, a task wishing to block other tasks from a resource first creates a full semaphore, and then takes the semaphore, making it unavailable to other tasks. When it is no longer needed, it the task gives the semaphore, making the resource available. A mutual exclusion semaphore must have matching “takes” and “gives”.

A binary semaphore can be viewed as a flag that is available (full) or unavailable (empty). When a task takes a binary semaphore, with semTake( ), the outcome depends on whether the semaphore is available (full) or unavailable (empty) at the time of the call. If the semaphore is available (full), the semaphore becomes unavailable (empty) and the task continues executing immediately. If the semaphore is unavailable (empty), the task is put on a queue of blocked tasks and enters a state of pending on the availability of the semaphore.

When a task gives a binary semaphore, using semGive( ), the outcome also depends on whether the semaphore is available (full) or unavailable (empty) at the time of the

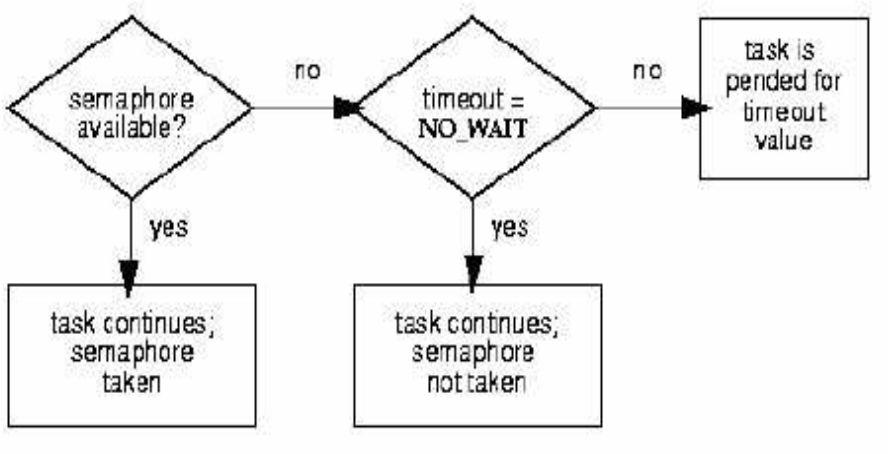

Figure 3.8 Taking a Semaphore

call. If the semaphore is already available (full), giving the semaphore has no effect at all. If the semaphore is unavailable (empty) and no task is waiting to take it, then the semaphore becomes available (full). If the semaphore is unavailable (empty) and one or more tasks are pending on its availability, then the first task in the queue of blocked tasks is unblocked, and the semaphore is left unavailable (empty).

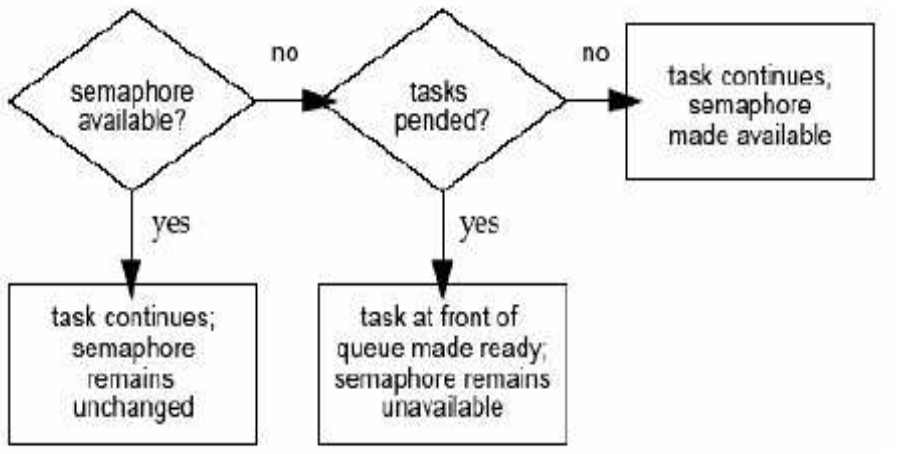

Figure 3.9 Giving a Semaphore

3.3.3 Message Queues

Closely related to the idea of semaphores is the concept of the message which passes data between tasks [7]. Messaging could be used to accomplish task interlocking as well, but setting up a message consumes more time than initializing a semaphore. Messaging is useful for passing variables between asynchronous tasks. While semaphores provide a high-speed mechanism for the synchronization and interlocking of tasks, often a higher-level mechanism is necessary to allow cooperating tasks to communicate with each other. In VxWorks, the primary intertask communication mechanism within a single CPU is message queues. Message queues allow a variable number of messages, each of variable length, to be queued. Tasks can send messages to a message queue, and tasks can receive messages from a message queue. Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction.

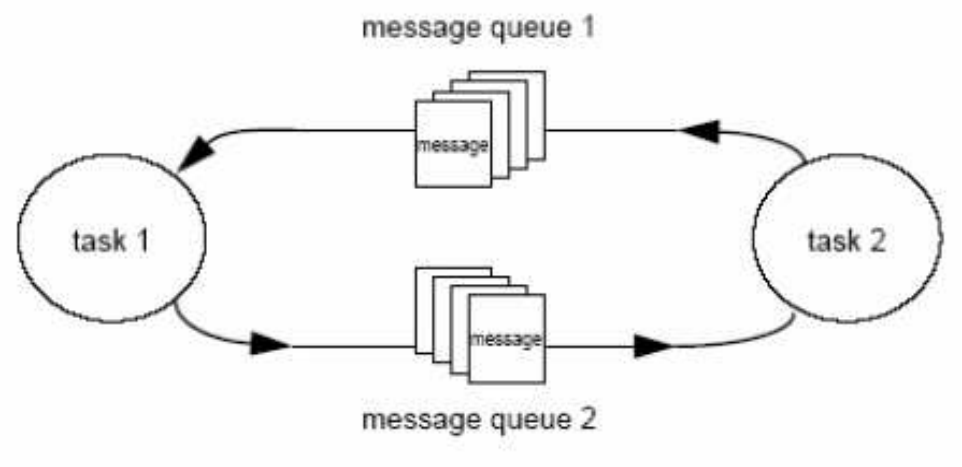

Figure 3.10 Full Duplex Communication Using Message Queues

3.3.4 Input/Output

In VxWorks, an input/output data stream is treated as a file regardless of the I/O device. File I/O can be organized by block or by character. Character devices include display terminals and external hardware devices such as A/D converters or real-time clocks. Block devices include local or network disk drives. File I/O can be real, as in the previous examples, or virtual as in the case of pipes and network sockets. Pipes enable tasks to communicate with each other. The pipe driver is called pipeDrv. If the device stream extends across a netowork, it becomes a socket. The stream socket is similar in concept to the Unix or Windows TCP/IP socket. For TCP/IP, VxWorks uses Berkeley Software Distribution version. 4.x Unix socket functions.

Table 3.1 File I/O Operations

create (const char *name, int flag)	Create a file
open (const char *name, int flags, int mode)	Open (and possibly create) a file
close (int fd)	Close a file
remove (const char *name)	Remove a file
read (int fd, char *buffer, size_t maxbytes)	Read an existing file
write (int fd, char *buffer, size_t nbytes)	Write to an existing file
ioctl (int fd, int function, int arg)	Perform control operations on a file

A file is handled by its file descriptor fd corresponding to a FILE structure in POSIX. For each kind of file, there is a different kind of driver which permits the following I/O operations, summarized in the following table 3.1. The string name denotes what kind of device the file is. When a file is created or opened, the I/O system searches through a list of device names, and physical file directories for at least a partial beginning match of the names and a file descriptor is returned if a match is established. If no match is found, a default device can be specified, or if no default is specified which is the more

common case, then the I/O system reports an error. Thereafter, the basic I/O operations specified previously are mapped to the specific file's I/O routines.

CHAPTER 4

THE APPROACH

We have developed a framework for tracing and replay of real-time concurrent programs in VxWorks. It is based on the work in [Ad91]. They have developed a language-based approach for replaying partial and total ordering based synchronization sequence deterministically in concurrent Ada programs. This approach is based on the order of the events synchronization.

The approach in [4] is extended to include the notion of time for the total ordering based real-time concurrent software developed in VxWorks. Also it is extended to control thread-switching to some extent to replay the real-time concurrent programs with more accuracy.

Using this approach, tracing and replay can be carried out into three stages:

1. Synchronization sequence definition
2. Synchronization sequence collection and
3. Synchronization sequence replay

4.1 Synchronization Sequence Definition

To carry out the language based deterministic execution of programs, we need to determine about what information is needed to be collected and the format of the information. This information must be defined carefully so that it is minimum and sufficient to repeat the deterministic execution and to collect some debugging information.

In order to define the synchronization information to be collected, first we should identify the events that may occur in the real-time concurrent programs execution and

which may lead to nondeterminism. These events can be either synchronization events for threads communication and data sharing or other thread related events that may change the interleavings of threads in the concurrent program under execution.

To determine this information, it is necessary to identify the sources of nondeterminism in the language. Here are some of the situations that may lead to nondeterminism.

1. Conditional synchronization: Depending on the result of the evaluation of some condition, the program flow may change. For example, depending on the value of some shared variable, different threads may follow different paths every time the program is executed.
2. Deadlock: Depending on how threads interact in the system and their progress, a deadlock situation might occur in the program.
3. Start of execution of a delay statement.
4. At the end of the execution of a delay statement, the program might exercise different delay alternatives or an exception might be generated.
5. Depending on the sequence in which threads may acquire some shared resource, the flow of the execution of the concurrent program may change.
6. If more than one thread is released due to some nonblocking operation, they may exercise different execution sequence every time.
7. The concurrent threads with the same priority may be scheduled in different pattern during each run of the program.
8. Request for some shared resource that is not satisfied within certain time limit.

Depending on the above and the other sources of nondeterminism in the system, we can identify the features in a particular language that are needed to be monitored to collect the required information for debugging and replay.

Most of the above scenarios for non determinism are due to intertask communication. So to collect the synchronization sequence, we need to monitor the intertask communication routines available in VxWorks.

Various intertask communication mechanisms available in VxWorks are:

- Shared memory, for simple sharing of data.
- Semaphores, for basic mutual exclusion and synchronization.
- Mutexes and condition variables for mutual exclusion and synchronization using POSIX interfaces.
- Message queues and pipes, for intertask message passing within a CPU.
- Sockets and remote procedure calls, for network-transparent intertask communication.
- Signals, for exception handling.

A program can be replayed deterministically if all these means of intertask communication are monitored and controlled to execute in the predefined order.

To replay the concurrent real-time programs, in addition to the order of the events in the reference execution sequence, time at which these events occurred is also of the significance.

Consider the message queues example for synchronization sequence collection: In VxWorks, the primary intertask communication mechanism within a single CPU is message queues. Message queues allow a variable number of messages, each of variable length, to be queued. Multiple tasks can send to and receive from the same message queue. The two basic operations for message queues are msgQSend and msgQReceive operation.

Task is blocked on msgQReceive if there is no message in the message queue and waits for another task to put a message. The task receives a message and it is unblocked in the order in which it started waiting for a message. That is the message queue is first in first out. In a message-queue based concurrent program, the execution sequence may

be different in each run of the program depending on the sequence in which tasks request for messages from the message queue.

The execution of a message queue based program may exercise different synchronization events given below:

- The execution of a msgQSend operation by a task
- The start of execution of a msgQReceive operation by a task
- The end of execution of a msgQReceive operation by a task

Since the msgQSend operation is a nonblocking operation, there is no need to separate the start and the end of the msgQReceive operation. To reproduce an event, we need to store some information about that event. For example, for a msgQSend event, the information we need to collect can be given as (T, MQ, M, E, S); where T is the task that executed the event, MQ is the id of the message queue in which message is to be sent, M is the message, E is the type of the event (i.e. msgQSend) and S is the time at which the event occurs. Similarly, the information about msgQReceive operation can be given as (T, MQ, MB, E, S); where T is the task that executed the event, MQ is the id of the message queue form which the task is waiting to receive a message, MB is the buffer to store the message, E is the type of the event (i.e. msgQReceive) and S is the time at which the event occurs.

Thus, the execution of a message queue based program can be characterized using the sequence in which these two operations take place and it can be denoted by ((T1,MQ1, M1, E1, S1),(T2,MQ2,M2,E2,S2),) where (Ti,MQi,Mi,Ei,Si) denotes the ith synchronization event in the sequence. Consider the message queue based scenario given below where four tasks are running concurrently and they perform send and receive operations on the same message queue:

```
Task1:
msgQSend (msgQ, msg1, …);
Task2:
msgQSend (msgQ, msg2, …);
Task3:
msgQReceive (msgQ, msgBuf1, …);
Task4:
msgQReceive (msgQ, msgBuf2, …);
```

Figure 4.1 A sample program for message queue operations

Depending on the order in which the above tasks execute send and receive events, one possible synchronization sequence can be given as:

$$((Task1, msgQ, msg1, msgQSend, s1), (Task3, msgQ, msgBuf1, msgQReceive, s2),$$

$$(Task2, msgQ, msg2, msgQSend, ts), (Task4, msgQ, msgBuf2, msgQReceive, s4))$$

The other possible synchronization sequence can be given as:

$$((Task1, msgQ, msg1, msgQSend, t1), (Task2, msgQ, msg2, msgQSend, t2),$$

$$(Task4, msgQ, msgBuf2, msgQReceive, t3), (Task3, msgQ, msgBuf1, msgQReceive, t4))$$

The information about a synchronization event involving a semaphore can be collected as:

$$(Task, semaphore, E, t)$$

where Task is the id of the task that is waiting for a semaphore, semaphore is the id of the semaphore, E is the type of the event (semTake or semGive) and t is the time at which this event occurred.

In addition to the intertask communication mechanisms, we also need to collect information about various task life-cycle events like create, activate, resume, restart, lock

and unlock. These events when executed produce task context switch. For example, when a new task is created, depending on the priority either the parent task continues executing after a context switch or the CPU control is given to the child task. The parent or the child task continues execution until another context switch occurs or the executing task requests a synchronization event execution. Because of the unpredictable task switching behavior of the program, some schedule latency might be introduced. If the time at which these task control routines are executes is controlled, we can replay the program more accurately in terms of time. These task control operations are handled in the same way as blocking operations in the case of the synchronization events.
According to the above discussion, the information about a taskSpawn() event can be represented as:

$$(TaskParent, pParent, TaskChild, pChild, t)$$

where TaskParent is the id of the task that created a child task, pParent is the priority of the parent task, TaskChild is the id of the newly created task, pChild is the priority of the newly created task and t is the time at which this event occurred. Also, some other parameters related to task stack and variables can be stored for further debugging aid.

A real-time concurrent program may execute a large number of synchronization and task control events and the size of the trace file may explode. To limit the size of the trace file and to allow more parallelism to threads while replay of the program according to the collected synchronization sequence, we should further reduce the information required to be collected for each event. Consider a semaphore based program given below in Figure 4.2. This program contains two threads and they share a variable x. Access to the shared variable x is mutually exclusive using a semaphore.

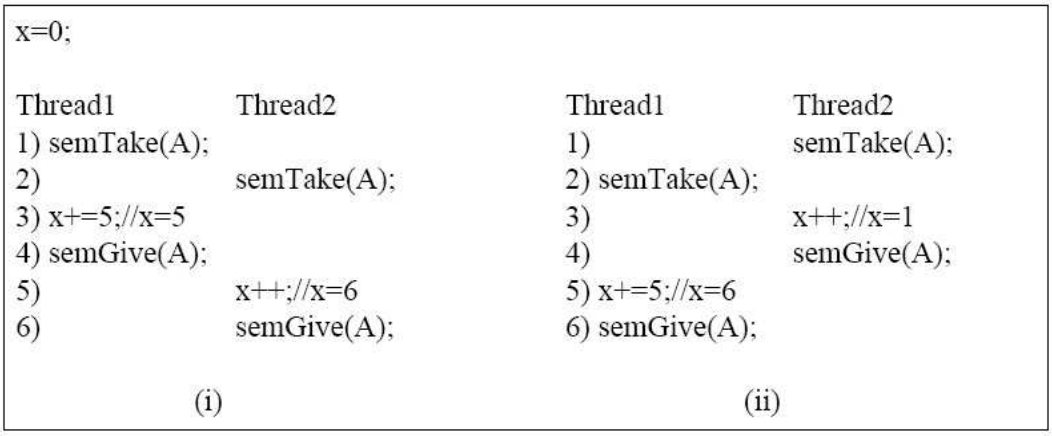

Figure 4.2 Sample interleavings of threads for semaphore events

If one thread requests for permission to access the shared variable using semTake() first, the other thread is blocked until the first thread releases the permit using semGive(). If threads request for permission using semTake() to access the shared variable in the order shown in (i), the value of x read by Thread1 is 5 whereas that read by Thread2 is 6. Now, if the threads interleave in a different way and they request for the permit in the order shown in (ii), the value of x read by Thread1 is 5 whereas that read by Thread2 is 1. These results show that the order in which the threads can access the shared variables depends on the order in which they request for permit using semTake(). The order in which the shared variable is accessed can be controlled if we force the threads to request for permission in the same order every time.

From the above discussion, it can be seen that the order in which semGive() operation is executed can be ignored. Similarly, for message queues, the order of execution of the non-blocking operation msgQSend() can be ignored. The synchronization sequence collected for the program in Figure 4.1 can be reduced to the one given below:

$(Task3, msgQ, msgBuf1, msgQReceive, s2), (Task4, msgQ, msgBuf2, msgQReceive, s4))$

Also, for total ordering based replay, if we control the order and time in which the tasks execute events irrespective of the types of the events, we can easily reproduce the program execution. So, the sequence above can further be reduced to the one given below:

$$((Task3, s2), (Task4, s4))$$

For task control events also the task id and the time are sufficient to replay that event.

4.2 Synchronization Sequence Collection

The synchronization sequence collection problem is addressed in this section.

It is much simpler than the synchronization sequence determination and the replay problem.API wrappers are developed for the synchronization constructs and the task control routines identified in the Synchronization Sequence Definition step above.

These API wrappers are developed to collect and write the synchronization sequence into the trace file. In addition to the synchronization sequence, some extra debugging information like priority of a thread, values of some variables, thread status, or information related to other threads, synchronization constructs or some application specific information can also be recorded.

To transform the original program, the API routines for which the synchronization sequence is to be collected are replaced with the corresponding API wrappers in the original program. These wrappers are instrumented for synchronization sequence collection. In addition to the instrumentation, these wrappers use original constructs for desired operations. Each time a synchronization object is accessed by a thread or a task control construct is executed, an entry for the corresponding thread id and the current value of the internal tick counter is made in the trace file. The trace file must be accessed exclusively by tasks and a task should not leave the wrapper routine before making a corresponding entry into the trace file. Also, the entry in the trace file should

be in the sequence in which the event is executed. Our assumption that the thread leaves the CPU control only when it is blocked on a synchronization operation like semTake(), msgQReceive() or after a task life-cycle routine taskLock() supports this requirement feasibility.

The instrumentation for information collection in the routine might affect the program behavior by introducing some delay and lead to a different execution sequence sometime. To keep the probe effect to minimum, it is necessary to decide carefully about what information is needed to be collected and it should be kept as minimum as possible. We can keep track of the time spent for information logging and subtract it from the timer tick value before leaving the wrapper to keep up with the original schedule.

The information tracing involves two approaches; one for blocking synchronization and task control routines and the other for task control routines that involve task switching immediately after execution. The general algorithm for an API wrapper for synchronization information collection for blocking routines like semTake(), msgQReceive() and taskLock() can be given as in figure 4.3.

```
returnType SynchronizationConstruct( required parameters, thread_id){
    status SynchronizationOperation;
    x=getTimerValue(); //gives internal timer value in terms of ticks
    record the status of the synchronization operation, the thread id, the current
    value of the internal timer and other debugging information into the trace
    file;
    y=getTimerValue();
    setTimerValue( getTimerValue()-(y-x));
    return;
}
```

Figure 4.3 Synchronization sequence collection for synchronization routines

With the blocking API instrumentation layer, we are able to monitor and log what blocking synchronization constructs are invoked. However, to record the time when a task control routine is executed, we need to be able to access the exact time when the task switch occurs. VxWorks provides several hooks, implemented as empty callback functions, such as a TaskSwitchHook[APPENDIX]. These hooks can be used for instrumentation purposes, making it possible to monitor and log sufficient information of each task switch in order to be able to reproduce it. The tracing algorithm for task life-cycle routines like taskSpawn(), taskActivate(), taskRestart() and taskResume() can be given as in figure 4.4

```
returnType TaskControlRoutine( required parameters){
      status TaskControlOperation;
      return;
}

Status taskSwitchHookRoutine(){
      x=getTimerValue(); //gives internal timer value in terms of ticks
      record the status of the task control operation, the thread id, the current value
      of the internal timer and other debugging information into the trace file;
      y=getTimerValue();
      setTimerValue( getTimerValue()-(y-x));
      return;
}
```

Figure 4.4 Synchronization sequence collection for task life-cycle routines

taskSwitchHookRoutine() is executed every time one of the task control routine is executed.

4.3 Synchronization Sequence Replay

The next step is to replay the program using the synchronization sequence defined and collected as described in the previous two sections. The trace file generated in the synchronization sequence collection phase is used to collect the sequence of thread ids

and times for events execution. The corresponding data structures are populated with thread ids and time in ticks. These data structures are used in the next run to replay the total ordering based sequence of events.

To replay the program, the wrapper API is instrumented with additional routines to control the execution of threads according to the collected synchronization sequence. The general algorithm for an API wrapper for synchronization information replay can be given as in figure 4.5

```
returnType SynchronizationConstructName( required parameters, thread_id){
        while( (seq[index] != thread_id) && (getTimerValue() < eventTime[index] ))
                wait();
        if(getTimerValue() > eventTime[index]){
                Print the number of ticks the task is delayed by (that is the value of
                        (getTimerValue() – eventTime[index]) in terms of ticks)
                status SynchronizationOperation;
        }
        else
                status SynchronizationOperation;
        index++;
        signal all other waiting threads
        return;
}
```

Figure 4.5 The general algorithm for Synchronization Sequence Replay

The thread is allowed to run only if it comes on or after the time it is scheduled and not if it comes before.If the thread comes before its scheduled time, according to the scenario shown in figure 4.6 and figure 4.7 below, there are chances that though the threads are executed in the same sequence, the end result might not be the same. In figure 4.6, the last value of x read by thread 1 is 6 whereas though followed is the same execution sequence, as shown in Figure 4.7, the last value of x is 2.

Now according to the scenario 2, thread 2 comes earlier than thread 1 and if it has higher priority than thread 1, under normal circumstances, it is granted execution and the end result changes. But using the wrapper API, thread 2 irrespective of other parameters that might affect, is not given the chance to execute before its noted execution time in the first run. So, thread 1 gets the same number of ticks to execute as in the first run before the context switch occurs with thread 2.

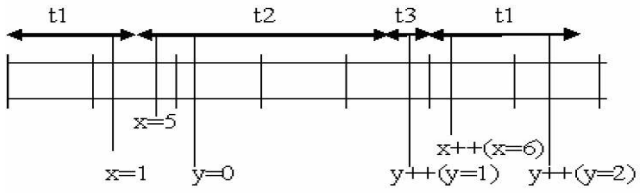

Figure 4.6 Reference schedule of tasks

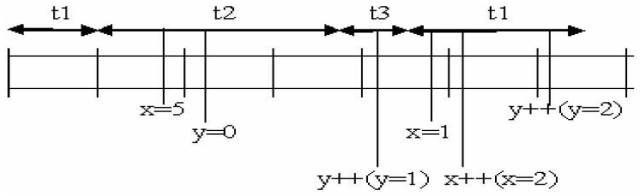

Figure 4.7 Another possible interleaving of tasks shown in figure 4.6

In case the thread comes later than its scheduled execution time, debugging information is produced. This information can contain the number of ticks the thread is delayed by, information and status of other threads, status and value of other shared variables etc. according to the problem to be analyzed and test.

According to our approach for replay, each thread has to request for a permit before executing the event. After the event is executed successfully, the thread releases

the permit to allow other threads in sequence to request for permit and execute events. The algorithms used for requesting and releasing the permits are given below:

In the algorithms' definitions, SYNC_SEQ is a feasible synchronization sequence. To demonstrate that how the algorithms work, let us assume that first i-1 operations are finished; that is index equals i and SYNC_SEQ[i] is j and TICK[index] is k. When a thread t where t does not equal to j enters REQUEST_PERMIT before time k, that is before the value of CURRENT_TICK becomes k, thread t is blocked at statement 7. When thread j enters REQUEST_PERMIT, it exits the procedure that is it receives the permit to perform the blocking operation. Thread j then executes the event following the REQUEST_PERMIT procedure.

After executing the event, the task t executes the RELEASE_PERMIT procedure. If the executed event is a blocking operation, like semTake(), msgQReceive() or taskLock(), the task immediately executes the RELEASE_PERMIT procedure following the event execution routine. In case of non-blocking routines like taskSpawn(), taskActivate(), taskReastart() or taskResume(), the RELEASE_PERMIT is executed inside the taskSwitchHookRoutine(). This routine is executed every time there is a task switch because of one of the above mentioned non-blocking events execution. The general wrapper routines are designed as shown in the Figure 4.10.

Inside RELEASE_PERMIT procedure, the task t increments index to i+1, and returns permit by executing either semGive(retry), if there are tasks blocked on semTake(retry) in REQUEST_PERMIT, or semGive(mutex), if no such tasks exist.

The semGive(retry) operation starts a cascaded wake-up of tasks blocked on semTake(retry). The first blocked task is awakened and removed form the queue to determine whterh its identifier is SYNC_SEQ[i+1] and whether CURRENT_TICK is greater than or equal to TICK[i+1]. If so, it exits REQUEST_PERMIT. Otherwise, it executes semGive(retry) to awaken the next task blocked on semTake(retry) and then blocks itself

```
(*global variables*)
Var
      SYNC_SEQ:array[1…max] of integer;
      TICK_SEQ:array[1…max] of integer;
      mutex     :semBinary:=SEM_FULL;   (*for mutual exclusion*)
      WAIT_COUNT:   INTEGER;     (*number of waiting tasks*)
      WAKE_COUNT:   INTEGER;     (*number of awakened tasks*)
      CURRENT_TICK: INTEGER;     (*current value of internal timer*)
      retry   :semBinary:=SEM_EMPTY;
      index   :INTEGER:=1;

procedure REQUEST_PERMIT(parameters, PROCESS_ID:integer);
1. begin(*REQUEST_PERMIT*)
2.            semTake(mutex);
3.    if(( PROCESS_ID <> SYNC_SEQ[index])
                  and (CURRENT_TICK < TICK_SEQ[index])) then
4.                begin(*wait*)
5.                 WAIT_COUNT:=WAIT_COUNT+1;
6.                 semGive(mutex);
7.                 semTake(retry);
8.                 while((PROCESS_ID <> SYNC_SEQ[index])
                              and (CURRENT_TICK < TICK_SEQ[index]))do
9.                   begin(*retry*)
10.                        WAKE_COUNT:=WAKE_COUNT+1;
11.                        if WAKE_COUNT < WAIT_COUNT then
12.                         semGive(retry);
13.                        else
14.                         semGive(mutex);
15.                         semTake(retry);
16.                  end;
17.                 WAIT_COUNT:=WAIT_COUNT-1;
18.               end;
19.           semGive(mutex);
20. end; (*REQUEST_PERMIT*)
```

Figure 4.8 REQUEST_PERMIT procedure

```
procedure RELEASE_PERMIT;
1. begin((*RELEASE_PERMIT*)
2.   semTake(mutex);
3.   INDEX:=INDEX+1;
4.   if WAIT_COUNT > 0 then
5.     begin (*cascaded wake up*)
6.       WAKE_COUNT:=0;
7.           semGive(retry);
8.         end (*cascaded wake up*)
9.   else
10.         semGive(mutex);
11. end; (*RELEASE_PERMIT*)
```

Figure 4.9 RELEASE_PERMIT procedures

again on semTake(retry). The cascaded wake-up continues until a blocked process is found whose identifier is SYNC_SEQ[i+1] and when CURRENT_TICK is greater than or equal to TICK[i+1]. If no such task is found after all blocked tasks are awakened exactly once that is if WAIT_COUNT equals WAKE_COUNT, the last task executes semGive(mutex) before blocking itself on semTake(retry).

A task blocked on semTake(retry) may be waiting at statement 7 or 15 in the REQUEST_PERMIT procedure. A blocked task that has not been awakened since entering REQUEST_PERMIT is waiting at statement 7. After being awakened once, a blocked task is waiting at statement 15. Here we assume that a task that is executing in the wrapper will relinquish the processor only if it is blocked during one of the blocking events mentioned above.

Tasks can call and enter the wrapper or the REQUEST_PERMIT procedure in any order, but they exit REQUEST_PERMIT in the order given by the SYNC_SEQ array and at time that is greater than or equal to the time given in the TICK array. After a process has exited REQUEST_PERMIT and executed the event in the order, it calls RELEASE_PERMIT to let the next event in the sequence to be executed.

```
SYNCHRONIZATIN CONSTRUCT WRAPPER {
//semTake(), msgQReceive(),taskLock() routines
        REQUEST_PERMIT
        SYNC_BLOCK_OPERATION
        RELEASE_PERMIT
}
TASK_CONTROL CONSTRUCT WRAPPER {
//taskSpawn(), taskActivate(),taskResume(), taskRestart() routines
        REQUEST_PERMIT
        TASK_CONTROL_OPERATION //TASK SWITCH
}
TASK_SWITCH_HOOK ROUTINE {
        RELEASE_PERMIT
}
```

Figure 4.10 API wrapper with REQUEST_PERMIT and RELEASE_PERMIT

4.4 The Experiment

In this chapter we look at an experiment which confirms the effectiveness of language-based framework for tracing and replay of real-time concurrent programs in VxWorks. In the example code, the taskMain() function creates various tasks. These tasks again create some tasks dynamically. All these tasks interact using two message queues msgQ1 and msgQ2. They share two variables; accesses to sharedVariable1 and sharedVariable2 are mutually exclusive using two binary semaphore variables sem1 and sem2.

During each run of this code, the input is the same; in terms of initial values of sharedVariable1 and sharedVariable2, both are initialized to 0. At the end of the execution, the values of these variables are monitored to see if their values read by the corresponding tasks are same during each run. Because of the non-determinism in the program, the values of these two variables and the corresponding two synchronization sequences observed are as given below in the Figure 4.11. The code for this example is given in the Figure 4.12.

Using the wrapper API functions to control the execution order and timing of events shown in the Figure 4.12 above, we are successfully able to replay the first synchronization sequence shown in the Figure 4.11. The comparison of the reference execution and the

One possible synchronization sequence and the values of shared variables:

((10,1),(1,2),(1,4),(5,5),(10,6),(4,7),(10,10),(2,13),(7,18),(7,20),(5,24),(5, 25),
(8,27),(3,30),(6,32),(6,33),(2,36))

Values of shared variables read by threads are:
sharedVariable1 : taskFour = 5; taskSeven=10;
sharedvariable2 : taskFive = 5; taskSix = 8;

The other possible synchronization sequence and the values of shared variables:

((10,1),(1,2),(1,4),(5,5),(10,6),(2,8),(10,12),(7,14),(7,20),(4,25),(5,26),(5,28),
(8,31),(3,34),(6,36),(6,37),(2,39))

Values of shared variables read by threads are:
sharedVariable1 : taskSeven = 5; taskFour=10;
sharedvariable2 : taskFive = 5; taskSix = 8;

Figure 4.11 Observed synchronization sequences and values of shared variables

replayed execution is shown in the Figure 3.13 below. This is an average case observation.

```
taskMain(): //id=10
            taskSpawn(taskOne); //create task one
            taskDelay(3); //introduce delay in terms of ticks
            taskSpawn(taskTwo); taskDelay(1);
            taskSpawn(taskThree);
taskOne(): //id=1
            taskSpawn(taskFour); taskDelay(1);
            taskSpawn(taskFive);      taskDelay(1);
            msgSend(msgQ2); //send message to the message queue
taskTwo():  //id=2
            taskSpawn(taskSix); taskDelay(5);
            msgSend(msgQ1); taskDelay(1);
            msgQSend(msgQ1); taskDelay(2);
            msgReceive(msgQ2); //receive message from the message queue
taskFour():  //id=4
            taskDelay(3);
            msgSend(msgQ2);
            semTake(sem1); //request for the semaphore
            sharedVariable1+=5;
            semGive(sem1); //release the semaphore
taskFive():  //id=5
            taskSpawn(taskSeven); taskDelay(10);
            semTake(sem2); taskDelay(5);
            sharedVariable2+=5;
            semGive(sem2);
            taskSpawn(taskEight);
taskThree():  //id=3
            msgReceive(msgQ1);
            msgSend(msgQ2);
taskSix():  //id=6
            msgReceive(msgQ1);
            semTake(sem2);
            sharedVariable2+=3;
            semGive(sem2);
taskSeven():  //id=7
            msgReceive(msgQ2);
            semTake(sem1);
            sharedVariable+=5;
            semGive(sem1);
taskEight():  //id=8
            msgReceive(msgQ2);
```

Figure 4.12 Events executed during the sample experiment execution

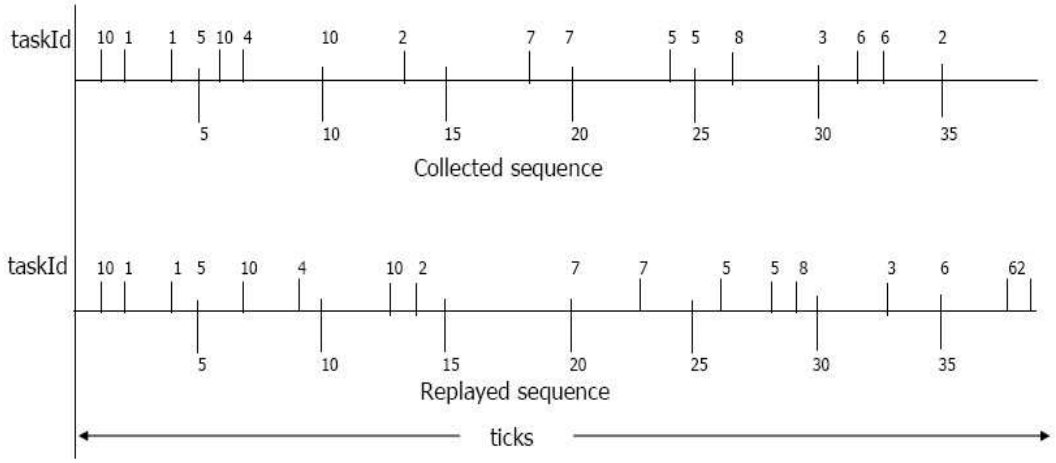

Figure 4.13 Comparison of reference and replayed events ordering and timing

CHAPTER 5

CONCLUSION

The work reported in this document confirms that it is easy to add some additional functionality in VxWorks API to allow the user to control the system scheduler, simply by monitoring the execution sequence and events timing at user level without any hardware support and low level interaction with the system. This undoubtedly will help the developers of VxWorks programs to perform the debugging operations more easily by replaying the erroneous behavior. While performing replay of the program, the framework also provides the elapsed time based statistics. This time information can be used for deadline monitoring, identifying the task which misses a deadline, and locating time-consuming code portions to support hand-tuning of tasks until a schedule becomes feasible. To provide this timing information, the execution speed of the application during replay becomes slower in average than the speed of the actual application without using the wrapper API. In contrast, conventional hardware simulators may provide the same information but are less portable and much slower. The environment facilitates the debugging of real-time programs when timing-related problems occur which have to be reproduced during debugging. Also, this approach of replaying the synchronization events and task life-cycle events can easily be ported for other languages as it uses very general functionality available in almost all languages to carry out the program tracing and replay.

CHAPTER 6

FUTURE WORK

We are successfully able to replay the real-time concurrent programs in VxWorks using our language-based framework. However this work addresses the nondeterminism contributed by synchronization constructs like semaphores, message queues and task life-cycle routine, there are many enhancements that would make the framework usable by all the real-time programs. The following would be the future enhancements:

- The approach can be enhanced for sporadic events and interrupts API. These are important features of real-time programs which are not addressed by this work but they can be implemented using the same approach.
- An external hardware timer/clock can be used to measure and trace timing parameters. System clock rate is the highest frequency we can achieve using software facilities provided by the VxWorks API. However, the same concept and implementation gives more accuracy with higher frequency measured using external timer/clock.
- More experiments can be performed using the developed API framework and statistics for the performance of the framework can be analyzed for its practical use. The approach can be demonstrated using a real-world application for better understanding and acceptance of the framework. We would also like to do a comparative study between the performance of the existing approaches and our work.

APPENDIX A

VxWorks API CONSTRUCTS

Semaphores

semLib - general semaphore library

This library provides semaphore routines.

SYNOPSIS

semGive( ) - give a semaphore

STATUS semGive

(

SEM_ID semId /* semaphore ID to give */

)

RETURNS OK, or ERROR if the semaphore ID is invalid.

The semGive( ) call relinquishes a specified semaphore, unblocking a pended task or making the semaphore available.

semTake( ) - take a semaphore STATUS semTake (SEM_ID semId, /* semaphore ID to take */ int timeout /* timeout in ticks */) RETURNS OK, or ERROR if the semaphore ID is invalid or the task timed out. The semTake( ) call acquires a specified semaphore, blocking the calling task or making the semaphore unavailable. All semaphore types support a timeout on the semTake( ) operation. The timeout is specified as the number of ticks to remain blocked on the semaphore. Timeouts of WAIT_FOREVER (-1) and NO_WAIT (0) indicate to wait indefinitely or not to wait at all.

Message Queues

msgQLib - message queue library

This library contains routines for creating and using message queues.

SYNOPSIS

msgQSend( ) - send a message to a message queue

STATUS msgQSend

(

MSG_Q_ID msgQId, /* message queue on which to send */

char * buffer, /* message to send */

UINT nBytes, /* length of message */

int timeout, /* ticks to wait */

int priority /* MSG_PRI_NORMAL or MSG_PRI_URGENT */

)

RETURNS OK or ERROR.

A task or interrupt service routine sends a message to a message queue with msgQSend(

).

msgQReceive( ) - receive a message from a message queue

int msgQReceive

(

MSG_Q_ID msgQId, /* message queue from which to receive */

char * buffer, /* buffer to receive message */

UINT maxNBytes, /* length of buffer */

int timeout /* ticks to wait */

)

RETURNS the number of bytes copied to buffer, or ERROR.

A task receives a message from a message queue with msgQReceive( ).

Task control

taskLib - task management library

This library provides the interface to the VxWorks task management facilities.

SYNOPSIS

taskSpawn( ) - spawn a task

int taskSpawn

(

char *name, /* name of new task (stored at pStackBase) */

int priority, /* priority of new task */

int options, /* task option word */

int stackSize, /* size (bytes) of stack needed plus name */

FUNCPTR entryPt, /* entry point of new task */

int arg1, /* 1st of 10 req'd task args to pass to func */

int arg2, int arg3, int arg4, int arg5, int arg6, int arg7, int arg8, int arg9, int arg10)

RETURNS task ID, or ERROR if memory is insufficient or the task cannot be created.

Tasks are created with the general-purpose routine taskSpawn( ). Task creation consists of the following:

Allocation of memory for the stack and task control block (WIND_TCB), initialization of the WIND_TCB, and activation of the WIND_TCB. Tasks in VxWorks execute in the most privileged state of the underlying architecture. There is no limit to the number of tasks created in VxWorks, as long as sufficient memory is available to satisfy allocation requirements.

taskActivate( ) - activate a task that has been initialized

STATUS taskActivate

(

int tid /* task ID of task to activate */

)

RETURNS OK, or ERROR if the task cannot be activated.

This routine activates tasks created by taskInit( ). Without activation, a task is ineligible for CPU allocation by the scheduler. The tid (task ID) argument is simply the address of the WIND_TCB for the task (the taskInit( ) pTcb argument), cast to an integer:

$$tid = (int)pTcb;$$

The taskSpawn( ) routine is built from taskActivate( ) and taskInit( ). Tasks created by taskSpawn( ) do not require explicit task activation.

taskResume( ) - resume a task

STATUS taskResume

(

int tid /* task ID of task to resume */

)

RETURNS OK, or ERROR if the task cannot be resumed.

This routine resumes a specified task. Suspension is cleared, and the task operates in the remaining state.

taskRestart( ) - restart a task

STATUS taskRestart

(

int tid /* task ID of task to restart */

)

RETURNS OK, or ERROR if the task ID is invalid or the task could not be restarted.

This routine "restarts" a task. The task is first terminated, and then reinitialized with the same ID, priority, options, original entry point, stack size, and parameters it had when it was terminated. Self-restarting of a calling task is performed by the exception task. The shell utilizes this routine to restart itself when aborted.

taskLock( ) - disable task rescheduling

STATUS taskLock(void)

RETURNS OK or ERROR.

This routine disables task context switching. The task that calls this routine will be the only task that is allowed to execute, unless the task explicitly gives up the CPU by making itself no longer ready. Typically this call is paired with taskUnlock( ); together they surround a critical section of code. These preemption locks are implemented with a counting variable that allows nested preemption locks. Preemption will not be unlocked until taskUnlock( ) has been called as many times as taskLock( ).A semTake( ) is preferable to taskLock( ) as a means of mutual exclusion, because preemption lock-outs add preemptive latency to the system.

taskUnlock( ) - enable task rescheduling

STATUS taskUnlock(void)

RETURNS OK or ERROR.

This routine decrements the preemption lock count. Typically this call is paired with taskLock( ) and concludes a critical section of code. Preemption will not be unlocked until taskUnlock( ) has been called as many times as taskLock( ). When the lock count is decremented to zero, any tasks that were eligible to preempt the current task will execute.

Ticks(Timer)

tickLib - clock tick support library

This library is the interface to the VxWorks kernel routines that announce a clock tick to the kernel, get the current time in ticks, and set the current time in ticks.

SYNOPSIS

tickSet( ) - set the value of the kernel's tick counter

void tickSet

(

ULONG ticks /* new time in ticks */

)

This routine sets the internal tick counter to a specified value in ticks. The new count will be reflected by tickGet( ), but will not change any delay fields or timeouts selected for any tasks. For example, if a task is delayed for ten ticks, and this routine is called to advance time, the delayed task will still be delayed until ten tickAnnounce( ) calls have been made.

tickGet( ) - get the value of the kernel's tick counter

ULONG tickGet (void)

RETURNS The most recent tickSet( ) value, plus all tickAnnounce( ) calls since.

This routine returns the current value of the tick counter. This value is set to zero at startup, incremented by tickAnnounce( ), and can be changed using tickSet( ).

tickAnnounce( ) - announce a clock tick to the kernel

void tickAnnounce (void)

This routine informs the kernel of the passing of time. It should be called from an interrupt service routine that is connected to the system clock. The most common frequencies are 60Hz or 100Hz. Frequencies in excess of 600Hz are an inefficient use of processor power because the system will spend most of its time advancing the clock. By default, this routine is called by usrClock( ) in usrConfig.c.

Task Hook

taskHookLib - task hook library

This library provides routines for adding extensions to the VxWorks tasking facility.

SYNOPSIS

taskCreateHookAdd( ) - add a routine to be called at every task create

STATUS taskCreateHookAdd

(

FUNCPTR createHook/* routine to be called when a task is created */

)

RETURNS OK, or ERROR if the table of task create routines is full.

This routine adds a specified routine to a list of routines that will be called whenever a task is created. The routine should be declared as follows:

void createHook

(

WIND_TCB *pNewTcb /* pointer to new task's TCB */

)

taskSwitchHookAdd( ) - add a routine to be called at every task switch

STATUS taskSwitchHookAdd

(

FUNCPTR switchHook /* routine to be called at every task switch */

)

RETURNS OK, or ERROR if the table of task switch routines is full.

This routine adds a specified routine to a list of routines that will be called at every task switch. The routine should be declared the same way as routine for taskCreate-HookAdd().

REFERENCES

[1] D. Pan and M. Linton, "Supporting reverse execution of parallel programs," *Proc. SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 1988.

[2] R. Netzer, "Optimal tracing and replay for debugging shared-memory parellel programs," *Proc. ACM/ONR Workshop on Parellel and Distributed Debugging*, 1993.

[3] Y. E.H.Paik and C.-W. Yoo, "A concurrent program debugging environment using real-time replay."

[4] R. H. C. K.-C. Tai and E. E. Obaid, "Debugging concurrent ada programs by deterministic execution," *IEEE Trans. Software Engineering*, vol. 17, 1991.

[5] J. C. Steve MacDonald and D. Novillo, "Deterministically executing concurrent programs for testing and debugging."

[6] A. P. H. Thane and D. Sundmark, "The asterix real-time kernel," *In Proceedings of the Industrial Session of the 13th EUROMICRO International Conference on Real-Time Systems*, 2001.

[7] W. Systems, "Vxworks 5.4 programmers guide," vol. First Edition.

BIOGRAPHICAL STATEMENT

Daxa Patel received her M.S. in Computer Science and Engineering degree from the University of Texas at Arlington in Dec 2006. She finished her undergraduate in Computer Engineering from the Atimiya Institue of Technology and Science, India in Jun 2004. Her research interests include testing of concurrent and real-time programs, object-oriented technology and use of computer science in enhancing quality of life.