THE FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

SERIAL I/O AND INTERRUPT SUPPORT FOR A POSIX THREADS KERNEL

By

ASHWIN GOYAL

A Project submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:

Fall Semester, 1994

The members of the Committee approve the project of Ashwin Goyal defended on September 30, 1994.

T. P. Baker
Professor Directing Project

E. W. Giering
Committee Member

D. B. Whalley
Committee Member

To my parents and grandparents...

Acknowledgements

I would like to express my gratitude to Dr. Baker, my major professor, for his guidance, support and patience during my work on this project. I would like to thank Frank Mueller for his invaluable help and Dr. Giering for his ever helpful suggestions during the course of this project. Also, I thank Dr. Baker and Dr. Whalley for being a constant source of inspiration throughout my two years at FSU.

Contents

List of Tables

List of Figures

Abstract

Pthreads (POSIX threads) is a draft IEEE standard interface for operating system services to support multiple threads of control within a single process. This report describes the design and implementation of extensions made to an existing Pthreads implementation on a bare Sun SPARCengine 1E, to support the Gnu Ada Runtime Library (GNARL). It shows how much better performance can be achieved by a standalone Pthreads library-based tasking implementation as compared to a user-level Pthreads library-based implementation. It also shows how a standalone Pthreads implementation can provide more predictable execution timing.

1 Introduction

The IEEE Portable Applications Standards Committee has been involved in specifying a standard interface for portable operating systems (POSIX) with various extensions. The IEEE Draft Standard P1003.1c [Tec94] specifies the interface for *threads*, which rely on shared memory and are lighter weight than processes. This report describes the extensions made to an existing POSIX threads (Pthreads) implementation on a bare Sun SPARCengine 1E[1] to support the Gnu Ada Runtime Library [BG94]. The extensions include a serial I/O driver and trap handling. The report confirms that a standalone kernel-level Pthreads implementation results in faster and more predictable performance than a library level implementation of Pthreads over a commercial Unix[2] operating system.

2 Background

2.1 Thread Concept

A *thread* is an independent sequential flow of control within a process. A process may consist of several threads, which can be created and terminated dynamically. Threads within a process share a single virtual address space and certain other system resources, but they maintain their own stacks, local variables and program counters. As for processes, they have different address spaces, file descriptors etc. Thus threads are considered to be lightweight, since the context of a thread is much less than that of a process and hence thread creation and termination, and context switches between threads, are much cheaper than the same operations on processes.

Threads are widely accepted as a computational building block for both uniprocessor and multiprocessor environments. In uniprocessor environments, the thread model simplifies the programming of asynchronous operations within a (heavyweight) process. A benefit of using threads in such environments is that context switches between threads incur less overhead than between processes. In multiprocessor environments, threads require shared-memory. In such environments they provide a low cost method to utilize multiple processors and to produce better throughput.

2.2 Relationship to Ada Tasks

The Ada programming language[U.S83] defines its own building block for concurrent programs, called *tasks*. Since Ada tasks are concurrent threads of control within a single program which share memmory (global objects), they are analogous to threads in a POSIX environment.

[1] Sun is a trademark of Sun Microsystems, Inc.; SPARC and SPARCstation are trademarks of SPARC International, Inc.

[2] Unix is a trademark of Unix Systems Laboratories, Inc.

If the underlying operating system supports the (C-language) POSIX threads interface, Ada tasking can be implemented using this interface, by mapping each task to a thread. In many respects the mapping between an Ada task and the corresponding thread is straightforward. For example, the optional scheduling policies that Pthreads support are compatible with the Ada Realtime Annex priority scheduling model. Other Ada features, such as the distinction between task creation and activation and the rules for task termination, are very different from their Pthreads counterparts, and have to be implemented almost entirely by the Ada runtime system.

The PART (POSIX Ada Real Time) project, at the Florida State University, is investigating the practicality of using POSIX threads to implement Ada tasking, especially in realtime applications. So far, a complete tasking implementation has been produced for the Ada 83 standard, using a library implementation of Pthreads Draft 6 layered over the Sun Unix operating system[GB92]. Work is under way to extend this system to the proposed new Ada 9X language standard [GMB93, Ada94].

To evaluate the suitability of a Pthreads-based Ada implementation for realtime applications, one must start out with a suitable realtime implementation of Pthreads. Experience with the FSU Unix-based Pthreads library implementation, and other Ada tasking implementations over ordinary Unix operating systems, makes it clear that acceptable realtime performance is not achievable by such an implementation. The problem is not so much a matter of efficiency as predictability. Most Unix implementations impose unpredictable delays on user processes, due to preemption by interrupt handlers and operating system processes. For an operating system to provide predictable timing of user threads, it must be designed with this objective in mind, from the hardware up.

3 Objectives and Scope

Earlier work provided a standalone Pthreads implementation on a bare Sun SPARCengine 1E [Rus93, BMR94], a processor designed for embedded realtime applications. The objective of this project is to extend the implementation to support the Gnu Ada Runtime Library (GNARL), which supports the proposed new Ada9X tasking features. A secondary objective is to evaluate the performance of this tasking library. The test programs include parts of the Ada Compiler Validation Capability (ACVC) and the Performance Issues Working Group (PIWG) benchmark test suites.

The Pthreads kernel is a RAM-only implementation. It neither supports virtual memory, nor a disk or any other secondary storage device. It supports only one process and this confirms to the *minimal realtime sytem profile* [Tec92].It is based on the earlier FSU library implementation[Mue93]. Reuse of code from this earlier implementation not only saved time, but also permitted direct performance comparisons of Ada runtime libraries based on the two implementations because the hardware is otherwise similar. Differences could be attributed to running on a bare machine versus as a layer over the Unix operating system.

The scope of the project was determined by the requirement for a standalone

Pthreads implementation to support the PART Ada tasking library to the point where it is possible to run the ACVC test suite. To achieve this, the following extensions to the original standalone Pthreads library were required:

- A Serial Input/Output driver: An efficient method to input and output data requested and generated, respectively, by applications.
- Trap handlers for appropriate synchronous signals: The Ada tasking library needs to be made aware of exception conditions through the use of synchronous signals.

4 Design

4.1 Serial Input/Output Driver

This section covers the design of the Serial I/O Driver. Writing the driver involved programming the communications chip available on the board, handling the interrupts generated by the chip and deciding on a synchronization protocol to be used by the reading and writing threads. Each of these are described in more detail in the following sections. The interface provided to user applications for input and output includes *sa_read* and *sa_write*, which are similar to the Unix *read* and *write* functions, respectively, and a partial implementation of the C library function *printf*.

4.1.1 Initializing the UART

The Sun SPARCengine 1E board uses a Zilog Z85C30 as a Universal Asynchronous Receiver Transmitter (UART). The Z85C30 Serial Communications Controller (SCC) Technical Manual [Adv88] states:

> The SCC is a dual channel, multiprotocol data communications peripheral designed for use with 8- and 16-bit microprocessors. It contains a variety of internal functions including on-chip baud rate generators, digital phase-lock loops and crystal oscillators, which dramatically reduce the need for external logic. It has 14 write registers and 7 read registers per channel. Using them, the SCC can be configured to handle all asynchronous formats regardless of data size, number of stop bits, or parity requirements. The SCC also accommodates all synchronous formats including character, byte and bit oriented protocols.

For the purposes of the Pthreads kernel, the SCC is programmed to be in an interrupt-driven asynchronous mode. The SCC is set with 8 bits per character, 2 stop bits and no parity bit. The baud rate generator is initialized to a rate of 9600. The SCC is programmed to interrupt on every character it receives, whenever the transmit buffer becomes empty, and on special receive conditions. A special receive condition occurs when there is an error, for example a receiver overrun error, which indicates that the receive FIFO buffer has overflowed. The interrupt vector, available in one of the read registers on the SCC (RR2), is used to determine the type of the interrupt generated by the SCC. The initialization sequence for the UART is explained in detail in appendix B.

4.1.2 Data Structures

The data structures used for the Input/Output driver include one buffer each for writing and for reading, associated with the UART.

The write buffer (*uart_write_buffer*) is a 256 byte long cyclic buffer. It holds the characters that will be put out by the UART in a cyclic sequential order. When the end of the buffer is reached the next character to be output is taken from the first location in the buffer, and so the cycle continues.

The read buffer (*uart_read_buffer*) is also a 256 byte long cyclic buffer and functions in a similar way, except that the characters are read into the buffer.

4.1.3 Protocol for reading and writing

Associated with reading and writing is a mutex for each operation, *read_mutex* and *write_mutex* respectively. They provide mutual exclusion for every write and read operation in every thread.

When a thread invokes a *write* function, the sequence of actions is (see Figure 1):

1. Lock the *write_mutex*. If the mutex is already locked, contend for the mutex until acquired.

2. Copy characters from the string to be output into the *uart_write_buffer*.

3. If the *uart_write_buffer* fills up before all the characters in the output string can be written into it, suspend the writing thread.

4. If thread is suspended, the UART interrupt handler outputs all the remaining characters in the output string and then wakes up the thread.

5. Unlock the *write_mutex*.

When a thread invokes a *read* function, the sequence of actions is (See Figure 1):

1. Lock the *read_mutex*. If the mutex is already locked, contend for the mutex until acquired.

2. Copy the required number of characters to be read, from the *uart_read_buffer* into the user-provided buffer.

3. If *uart_read_buffer* becomes empty before the required characters are copied, suspend the reading thread.

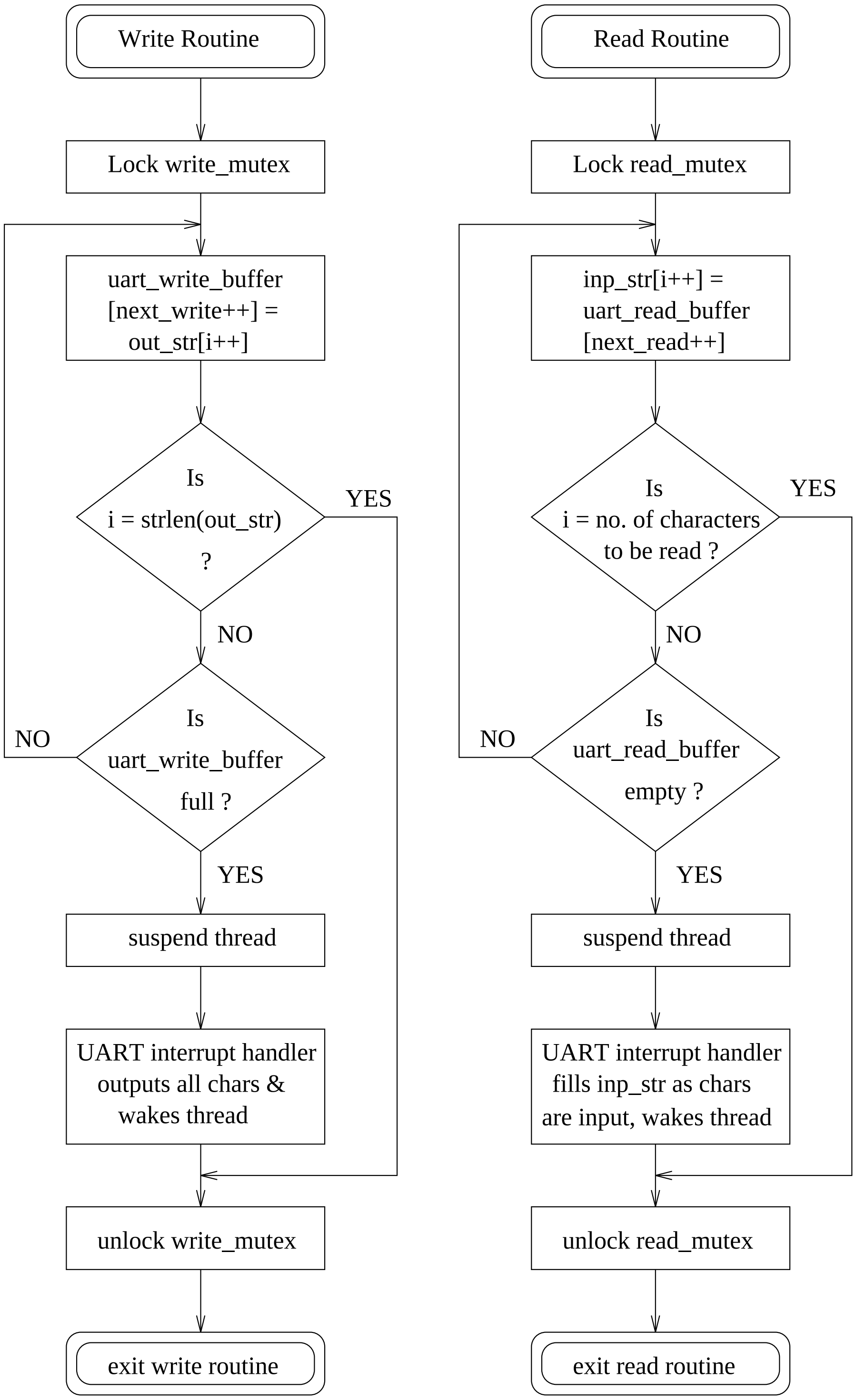

Figure 1: Write and Read Routine

4. If the thread is suspended, the UART interrupt handler fills the user-provided buffer with the remaining number of characters as the characters come in, then wakes up the thread.

5. Unlock the *read_mutex*.

4.1.4 UART Interrupt Handler

This section presents a overview of the UART interrupt handler. On a UART interrupt, the sequence of actions is (See Figure 2):

1. Determine the type of interrupt.

2. If it is a write interrupt

   (a) Output next character from the *uart_write_buffer*.

   (b) If there is a thread suspended on write

      i. Copy the next character from the output string into the *uart_write_buffer*.

      ii. If all the characters required to be written by the thread have been copied into the *uart_write_buffer*, wake up the thread.

   (c) Return from handler

3. If it is a read interrupt

   (a) Read input character into the *uart_read_buffer*.

   (b) If there is a thread suspended on read

      i. Copy the input character into the user-provided buffer.

      ii. If all the characters required by the thread have been input, wake up the thread.

   (c) Return from handler.

4. If it is an error interrupt

   (a) Reset the error condition

   (b) Return from handler.

4.2 Trap Handling

This section reviews the work done to implement trap handlers for exceptions like data access error, memory address not aligned, privileged instruction, etc.

In general, on an exception, the handler might do some exception specific processing and then generate an appropriate synchronous signal. Pthreads uses synchronous signals to indicate error conditions. These error conditions generate traps on the SPARC and the trap handlers determine the signal to be generated. The typical steps involved in writing a trap handler are:

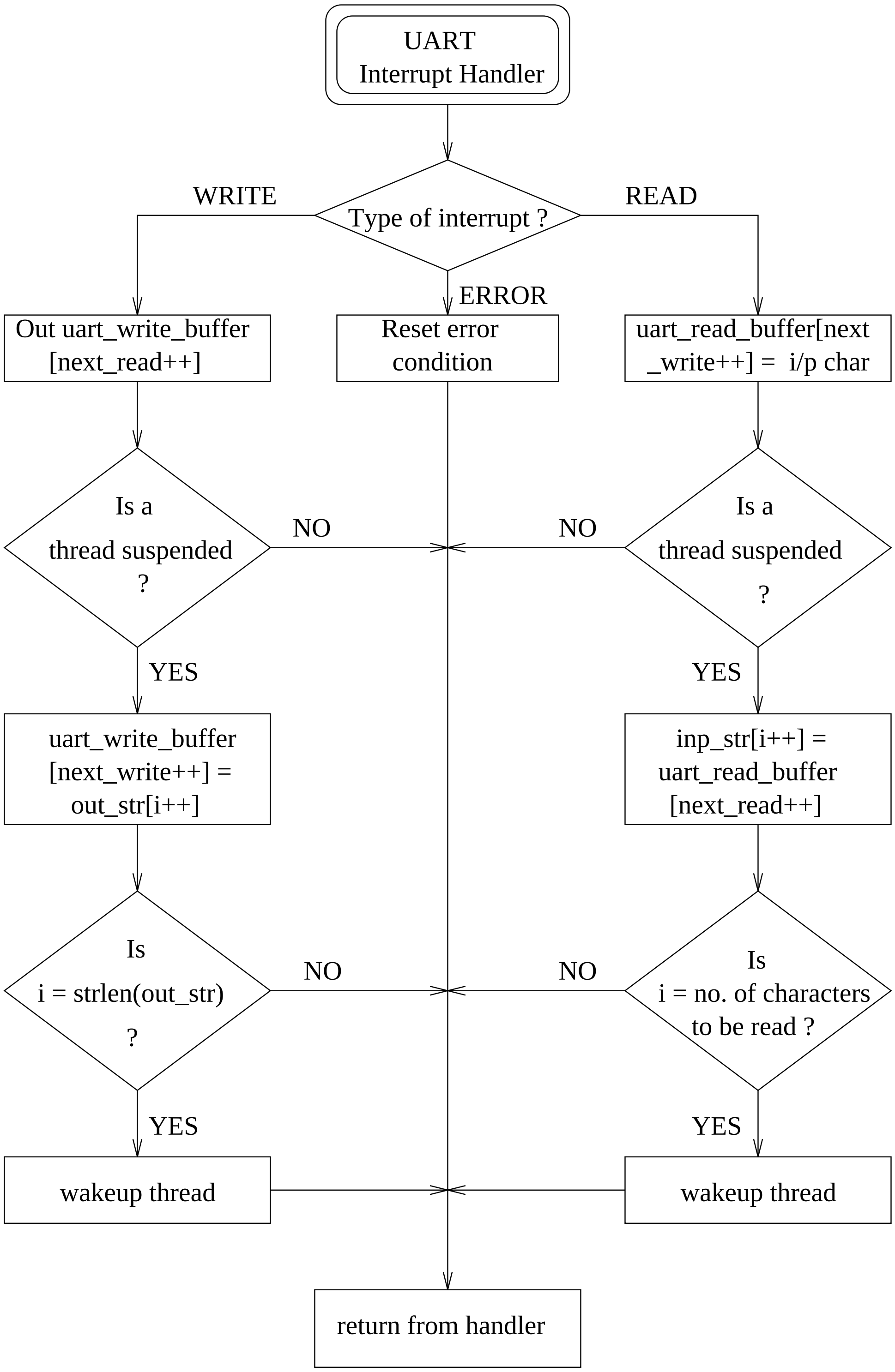

Figure 2: UART Interrupt Handler

1. Provide the handler routine. The routine determines the type of the signal that needs to be generated and adds it to the signal mask of the thread that caused the exception. The typical sequence of actions taken by such a handler is given in Figure 3.

2. Write initialization code to create an entry in the trap table and to enable the interrupt. The trap table maps traps to trap handlers. Whenever that trap occurs, the processor causes the execution of the handler.

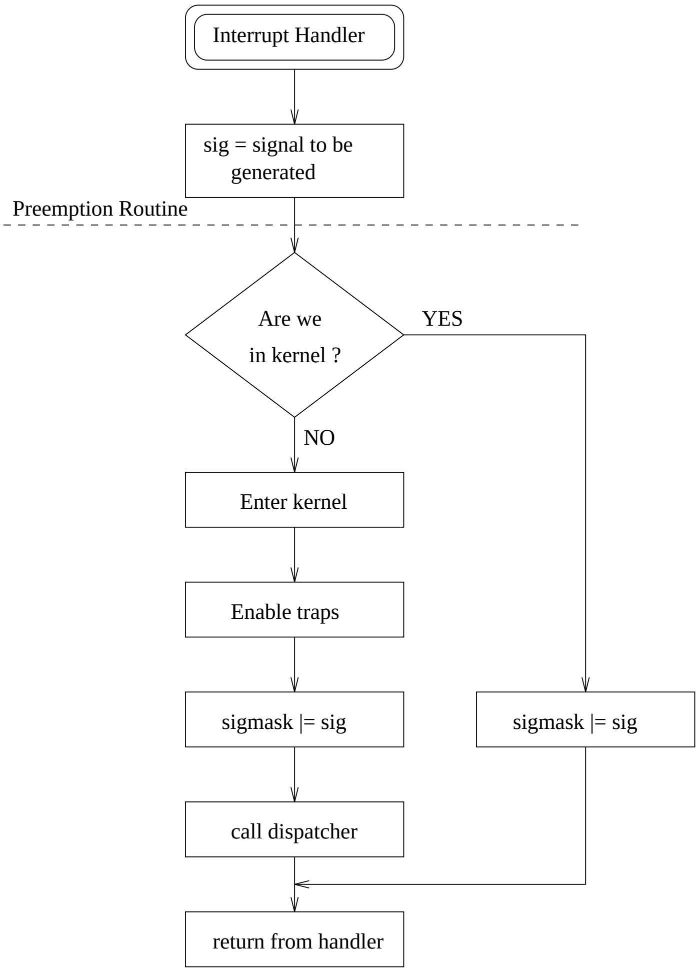

Figure 3: Generic Interrupt Handler

Here, it should be noted that in our kernel, thread preemption is supported through a common routine[MRB94]. Control can be transferred to this code from any interrupt handler. The semantics of the routine depend on whether user code or kernel code was interrupted.

If user code of a thread is interrupted and the interrupt handler determines that a scheduling action needs to be performed, then control is transferred to the preemption routine. This routine enters the kernel (by setting a flag) and creates a new stack frame on top of the interrupt handler. It then raises a signal and calls a routine which performs the scheduling action, followed by a call to the dispatcher. The current thread may lose the processor at this point. When the thread regains control of the processor it will resume its execution by returning from the interrupt handler.

If a thread is interrupted while in the kernel, the interrupt handler transfers control to the preemption routine which does not preempt the current thread. Instead it raises a process-level signal corresponding to the interrupt. The scheduling effect of the interrupt is deferred. The execution will then resume by returning from the interrupt handler, but will still be inside the kernel. Later, when control leaves the the kernel, the process signal mask is examined and pending signals are handled before the dispatcher attempts to switch contexts.

The mapping between traps and the generated signals is given in Table 1.

5 Measurements and Evaluation

The source code of our standalone Pthreads kernel consists of approximately 6500 lines of code, of which approximately 1800 lines are new code. Out of the remaining 4700 lines of code, about 2500 lines are reused from the user-level library implementation of Pthreads. The core image of the kernel is about 122 kilobytes, as compared to 1012 kilobytes for the full Sun Unix kernel.

The code was tested for both functionality and performance.

5.1 Functional Testing

The functionality tests included about 25 bare-machine tests that were used to test the previous implementation of the kernel [Rus93], additional tests written to check the extensions to the kernel, and a subset of the Ada Compiler Validation Capability (ACVC) test suite.

The features tested by the bare-machine tests include: thread management, including creation, termination, join, detach; priority scheduling; mutexes, with and without priority ceilings; condition variables; timed condition wait; thread-specific data; setjmp/longjmp; signal handlers; cancellation and cleanup handlers.

The new features tested include write, read, and exception handlers. These new tests include:

- *write_test:* This is a simple program that tests the output routine *sa_write.*
- *read_test:* This is a simple program that tests the input routine *sa_read.*
- *io_test1:* This program creates two threads that read and write interactively. It also tests the cyclic buffer.

Table 1: Mapping of traps to signals

Trap Type	Interrupt Level	Signal Generated	Signal Code
instruction/data access exception	5, 13	SIGSEGV	*segv_prot*
privileged instruction	6	SIGILL	*ill_privinstr_fault*
illegal instruction	7	SIGILL	*ill_illinstr_fault*
memory address not aligned	10	SIGBUS	*bus_align*
floating point exception	11	SIGILL	*ill_illinstr_fault* if unimplemented FP operation
		SIGBUS	*bus_align* if invalid FP register
		SIGFPE	*fpe_fltinex_trap* if unfinished FP operation *fpe_fltdiv_trap* if divide by zero error *fpe_fltinex_trap* if inexact result *fpe_fltund_trap* if underflow *fpe_fltovf_trap* if overflow *fpe_fptoperr* if invalid operand
integer divide by zero	16	SIGFPE	*fpe_intdiv_trap*
integer overflow	16	SIGFPE	*fpe_intovf_trap*

- *io_test2:* This program creates three threads that read and write. It tests the synchronization protocol used by the reading and writing threads. It also tests the *printf* routine.
- *acc_exp:* This program is used along with *gen_exp.c* to test various interrupt handlers.
- *fpu_test1:* This program tests the floating point divide by zero exception handler.
- *fpu_test2:* This program tests the integer divide by zero exception handler.
- *stack_overflow:* This program tests the stack overflow handling mechanism provided by our standalone Pthreads kernel.

The ACVC test suite is used by Ada implementors to check conformance of their compilers to the Ada language standard. The Ada9X compiler we use is the Gnu NYU Ada Translator (GNAT), which is being implemented at New York University [Uni]. For the PART project at FSU, the ACVC tests of interest are those that test Ada tasking and synchronization. Out of these tests, the ones that the current user-level Pthreads library-based tasking implementation passes were run on our standalone Pthreads library-based tasking implementation. Our tasking implementation successfully passed the tests. Some of the features that were tested are:

- Simple task creation and termination.
- Task elaboration.
- Task activation and execution.
- Passing tasks as IN or IN OUT parameters.
- Task completion caused by exception during its activation.
- Task completion caused by exception during its execution - with and without exception handler.
- Entry calls.
- Rendezvous.
- Subprogram/Entry overloading.
- Conditional entry call.
- Abortion.

The Pthreads features used by the ACVC tests include: thread creation, thread termination, priority scheduling, stack resource protocol, mutexes, condition variables, timed condition wait, and thread specific data. These tests use the write operation for output.

5.2 Performance Testing

Performance was tested using 11 tests which are a part of the PIWG test suite which is used to test the performance of Ada compilers.

Table 2 shows the measurements obtained. The timings for the Unix-based version were variable depending on how heavily loaded the machine was at that time. The table shows the range of the timing results. Using the measurements we compare the standalone Pthreads library-based tasking implementation and the one based on the user-level Pthreads library over Unix. The performance improvement is calculated using the best results obtained for the Unix-based version. The average performance improvement is 12.67 %.

Briefly, the performance metrics include:

- Task entry call and return time measurement:
  - *t000001:* 1 task, 1 entry, task inside procedure, no select.
  - *t000002:* 1 task, 1 entry, task in a package, no select.
  - *t000003:* 2 task, 1 entry per task, tasks in a package, no select.
  - *t000004:* 1 task, 2 entries, task in a package, select.
  - *t000005:* 10 tasks, 1 entry per task, tasks in a package, no select.
  - *t000006:* 1 task, 10 entries, task in a package, no select.
- Minimum rendezvous, entry call and return time measurement:
  - *t000007:* 1 task, 1 entry, no select
- Average time to pass an integer:
  - *t000008:* Average time to pass an integer from a producer task to a consumer task through a buffer task.
- Task create and terminate measurement:
  - *c000001:* 1 task, no entries, task in a procedure, using a task type in a package.
  - *c000002:* 1 task, no entries, task in a procedure, task defined and used in procedure.
  - *c000003:* 1 task, no entries, task is in a declare block in main procedure.

Table 2: Performance Measurements

Tests	Timings(μsecs) Standalone Implementation	Unix-based Implementation	Number of Iterations	Percentage Performance Improvement
t000001	1058	1240 - 1370	6400	14.68
t000002	1063	1220 - 1340	6400	12.87
t000003	1037	1220 - 1335	6400	15.0
t000004	1225	1430 - 1530	3200	14.34
t000005	1022	1280 - 1408	1000	20.16
t000006	1716	2040 - 2193	1000	15.88
t000007	557	661 - 724	3200	15.73
t000008	2233	2740 - 2960	800	18.50
c000001	6123	6360 - 6920	800	3.73
c000002	6153	6410 - 6890	800	4.01
c000003	6104	6390 - 6901	800	4.48

6 Conclusions and Future Work

We have successfully ported an Ada9x tasking library onto a standalone Pthreads library. We tested the performance of the standalone Pthreads library-based tasking implementation, and compared it to the user-level Pthreads library-based implementation. As expected, we found that the standalone version has better performance. Table 2 shows that the standalone version also has better execution time predictability. We expect the standalone version to continue to be used for future testing of GNARL.

A part of the performance and execution time predictability improvement is due to the fact that our standalone Pthreads kernel is a RAM-only implementation. It was designed from the start without virtual memory support, thereby avoiding time delays due to page faults. Another factor for better performance is that the algorithm used for saving register windows to memory is faster than the one used by SunOS 4.1.3 [Rus93].

6.1 Future Work

The FSU standalone Pthreads library implementation can be extended in several directions. It can possibly be extended to a more complete OS kernel or to a more stripped down version. These and a few more possibilities are briefly explored in the following paragraphs.

To implement a more complete OS kernel, memory protection for kernel data structures, virtual memory management and faster communication media would need to be implemented. For the latter, an ethernet driver would be needed.

The present implementation can be modified to a more stripped down version. It would involve making Pthreads operations async-unsafe whereas the current implementation is async-safe. This might lead to a faster implementation.

The implementation can also be redesigned to support multiprocessors. Several changes would have to made to the kernel. Most notably, the monolithic kernel structure should probably be changed to a multi-threaded one, with fine-grain locking of data structures to minimize the probability of contention between several threads running under kernel mode. Otherwise, the advantage of a multiprocessor would not be fully exploited.

A programming environment for threads should also provide debugging facilities with support for multi-threading[CB90]. Right now, the only debugging information available is through the Boot PROM Monitor[SUN91], which is at the machine language level. Some support should be provided that would enable users to debug at the source code level and would also provide some support for multi-thread debugging.

A Building the Kernel and the Standalone Programs

This appendix explains how to compile the standalone SPARC Pthreads kernel, and how to compile, link and execute the test programs. It is intended as a guideline for the person who wishes to do further development and testing of the kernel.

A.1 Hardware

The machine used for this work is a Sun SPARCengine 1E with 4 MB of main memory. The hostname of this machine is *midas*. The console of *midas* is connected to the serial port A of a workstation, whose hostname is *alpha*.

A.2 Software

The sources consist of three parts: Pthreads sources, kernel sources and C library sources. Pthreads sources are in the directory *threads/src* under my home directory (*/alpha1/goyal*), the kernel sources are in *threads/midas/src/threadsrc* and the C library sources are in the directory *threads/midas/src/libsrc.* All the directories contain a local Makefile. The Makefiles specify how to compile the sources. The object files are placed in the directory *threads/midas/lib/objs.* The linkable library is created in the directory *threads/midas/lib* using a Makefile in that directory. All the test programs are in the directory *threads/midas/tests.* They are compiled using a local *Makefile.* The test program executable is copied into */home/midasroot/tests* so that it can be run using the Open Boot PROM[3]. *Makefile*s have been commented to reflect the meaning

[3]Open Boot PROM is a trademark of Sun Microsystems, Inc.

of compile flags and compiling/loading options.

A.2.1 Compiling the Kernel and the Standalone Programs

Here follows a session that explains how to compile the standalone library, and compile and link an executable program using this library. While compiling the Pthreads sources, make sure that the correct CFLAGS option is set in the *Makefile.*

---

alpha% pwd
/alpha1/goyal
alpha% cd threads/src
alpha% make

```
cc -I../include -I/home/midasusr/include -g -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE -c pthread.c
cc -I../include -I/home/midasusr/include -g -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE -c stack.c
cc -I../include -I/home/midasusr/include -g -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE -c mutex.c
cc -I../include -I/home/midasusr/include -g -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE -c cond.c
cc -I../include -I/home/midasusr/include -g -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE -c pthread_init.c
cc -I../include -I/home/midasusr/include -g -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE -o get_offsets get_offsets.c
./get_offsets > pthread_offsets.h
cc -I../include -I/home/midasusr/include -g -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE -c signal.c
as -I../include -I/home/midasusr/include -P -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE pthread_sched.S -o pthread_sched.o
as -I../include -I/home/midasusr/include -P -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE pthread_aux.S -o pthread_aux.o
as -I../include -I/home/midasusr/include -P -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
```

```
-DSRP -DC_INTERFACE pthread_opt.S -o pthread_opt.o
cc -I../include -I/home/midasusr/include -g -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE -c queue.c
cc -I../include -I/home/midasusr/include -g -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE -c io.c
cc -I../include -I/home/midasusr/include -g -DASM_SETJMP \
-DNOERR_CHECK -DSTAND_ALONE -DSTACK_CHECK -DSIGNAL_STACK \
-DSRP -DC_INTERFACE -c pthread_disp.c
if (test -x /usr/bin/ranlib) ; \
        then \
  ar ru ../lib/libpthreads.a *.o ; \
          /usr/bin/ranlib ../lib/libpthreads.a ; \
        else \
  ar ru ../lib/libpthreads.a `lorder *.o | tsort` ; \
        fi
ar: creating ../lib/libpthreads.a
```

alpha% cd ../midas/src/threadsrc/
alpha% make

```
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP  -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/fast_ops.o fast_ops.S
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP  -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/malloc.o malloc.c
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP  -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/lock_page.o lock_page.S
cc -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP  -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -o get_pte get_pte.c
get_pte > dev_pte.h
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP  -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/locore.o locore.S
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/clock.o clock.c
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP  -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/write.o write.c
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
```

```
-DSRP -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/uart_routines.o \
uart_routines.c
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -o get_sa_pthread_offsets \
get_sa_pthread_offsets.c
./get_sa_pthread_offsets > sa_pthread_offsets.h
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/uart_enable.o \
uart_enable.S
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/select_isr.o select_isr.c
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/exception_handlers.o \
exception_handlers.S
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/mprotect.o mprotect.c
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/mprot.o mprot.S
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/exception.o exception.c
cc -O3 -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSRP -DNOERR_CHECK -DC_INTERFACE -DSTAND_ALONE  -DSTACK_CHECK \
 -DSIGNAL_STACK -c -o ../../lib/objs/gnarl_clock_gettime.o \
gnarl_clock_gettime.c
```

alpha% cd ../libsrc/
alpha% pwd
/alpha1/goyal/threads/midas/src/libsrc
alpha% make

```
cc -O3 -I/home/midasusr/include -c -o ../../lib/objs/stdio.o stdio.c
cc -O3 -I/home/midasusr/include -c -o ../../lib/objs/uart.o uart.S
cc -O3 -I/home/midasusr/include -c -o ../../lib/objs/div.o div.S
cc -O3 -I/home/midasusr/include -c -o ../../lib/objs/rem.o rem.S
cc -O3 -I/home/midasusr/include -c -o ../../lib/objs/multiply.o \
multiply.S
cc -O3 -I/home/midasusr/include -c -o ../../lib/objs/string.o string.c
```

```
cc -O3 -I/home/midasusr/include -c -o ../../lib/objs/new_uart.o \
new_uart.S
```

alpha% cd ../../lib
alpha% make

```
cp /alpha1/goyal/threads/src/*.o objs
rm -f libpthreads.a
ar ru libpthreads.a objs/locore.o objs/*.o
ar: filename exception_handlers.o truncated to exception_handl
ar: filename gnarl_clock_gettime.o truncated to gnarl_clock_get
ar: creating libpthreads.a
ranlib libpthreads.a
```

alpha% cd ../tests
alpha% make specific

```
cc -I/alpha1/goyal/threads/include -I/home/midasusr/include \
-DSTAND_ALONE -c specific.c
ld -o specific -e _start -n -p -X -T 4000 ../lib/libpthreads.a \
specific.o ../lib/libpthreads.a
cp specific /home/midasroot/tests/a.out
sync
```

A.2.2 Executing a Standalone Program

To run a standalone program, the Open Boot PROM is used from *midas'* console. The following is a session that explains how to run the standalone program (*specific*) compiled earlier, which is now present in the root directory of *midas* as /tests/a.out. Start two sessions of kermit, one for the PROM monitor and one for input/output. Since our standalone Pthreads kernel programs the UART to use its channel B for input/output, use the serial line B on *alpha* for input/output.

In kermit, the escape character is Ctrl-\. Use

Ctrl-\ b to send a BREAK

Ctrl-\ c to disconnect

---

alpha% kermit

```
C-Kermit, 4E(070) 29 Jan 88, 4.2 BSD
Type ? for help
C-Kermit>set line /dev/ttya
Warning, write access to lock directory denied
C-Kermit>set baud 9600
/dev/ttya: 9600 baud
C-Kermit>c
Connecting thru /dev/ttya, speed 9600.
The escape character is CTRL-\ (28).
Type the escape character followed by C to get back,
or followed by ? to see other options.
```

(Now, break into the PROM monitor using the escape sequence Ctrl-\ b. Also, start the second session of kermit before proceeding from this point. Instructions for starting the second sessions are given following this text.)

```
Type b (boot), c (continue), or n (new command mode)
>n
Type  help  for more information
ok boot le()/tests/a.out -h
Booting from: le(0,0,0)/tests/a.out -h
21a00 Using IP Address 128.186.121.47 = 80BA792F
hostname: midas
domainname: rts.fsu.edu
server name 'alpha'
root pathname '/home/midasroot'
root on alpha:/home/midasroot fstype nfs
Size: 57344+1704+1195368 bytes
Type  'go' to resume
ok go
ok
```

The kermit session on serial line B can be started as follows.

---

alpha% kermit

```
C-Kermit, 4E(070) 29 Jan 88, 4.2 BSD
Type ? for help
C-Kermit>set line /dev/ttyb
Warning, write access to lock directory denied
```

```
C-Kermit>set baud 9600
/dev/ttyb: 9600 baud
C-Kermit>c
Connecting thru /dev/ttyb, speed 9600.
The escape character is CTRL-\ (28).
Type the escape character followed by C to get back,
or followed by ? to see other options.
```

(Now, nothing more needs to be done. This screen will be used for input and output by the kernel.)

```
create thread 1
pthread_self = 77568
mutex is locked, execute 1 cleanup handler
could destroy 77568 now, arg 0
thread done.
could destroy 77568 now, arg -65793
create thread 2
pthread_self = 78240
kill thread 78240
could destroy 78240 now, arg 0
could destroy 78240 now, arg 0
could destroy 78240 now, arg 78240
create thread 3
pthread_self = 78912
kill thread 78912
unlocking mutex now
could destroy 78912 now, arg 78912
kill thread 78912
could destroy 76896 now, arg 76896
```

B Initialization sequence for the UART

This appendix explains the initialization sequence for the UART in detail. This sequence was adapted from the sequence which appears in Chapter 7 (SCC Application Notes, page 7-7) of the Z85C30 Serial Communications Controller Technical Manual [Adv88]. We initialize channel B of the UART by sending out the bytes sequence given in table 3 to the channel B control register. The formula used to arrive at the time

Table 3: Initialization sequence for the UART

Register	Value	Comments
0x9	0x40	Reset Channel B
0x4	0x4C	x16 clock, 2 stop bits, no parity
0x2	0x0	Initialize interrupt vector register
0x3	0xC0	Rx 8 bits/character, Rx disabled
0x5	0x60	Tx 8 bits/character, Tx disabled
0x9	0x0	Interrupts disabled
0xA	0x0	NRZ mode
0xB	0x56	Tx and Rx clocks come from Baud Rate Generator(BRG) output, BRG output signal is available at the TRxC pin of SCC
0xC	0xE	Time constant low = 14. The method used to arrive at this figure is explained in the text
0xD	0x0	Time constant high = 0
0xE	0x7	BRG enable
0x3	0xC1	Rx enable
0x5	0x68	Tx enable
0x1	0x12	Rx interrupt on all characters and Tx int enable
0x9	0x08	Master Interrupt Enable

constant for the baud rate generator is as follows.

$$Time\ Constant = \left[\frac{Clock\ Frequency}{2 * Clock\ Mode * Baud\ Rate}\right] - 2$$

$$Time\ Constant = \left[\frac{4915200}{2 * 16 * 9600}\right] - 2 \ = \ 14$$

The formula appears on page 5-3 of the SCC Technical Manual [Adv88]. The figures appear on page 68 of the SPARCengine User's manual [SUN90].

References

[Ada94] Ada 9X Mapping/Revision Team. *Ada 9X Reference Manual: Draft Version 5.0*, June 1994. Available by anonymous FTP from ajpo.sei.cmu.edu.

[Adv88] Advanced Micro Devices, Inc. *Z85C30 Serial Communications Controller Technical Manual*, 1988.

[BG94] T.P. Baker and E.W. Giering. PART/GNARL interface definition. Technical report, Florida State University, July 1994. Available by anonymous FTP from ftp.cs.fsu.edu.

[BMR94] T. P. Baker, F. Mueller, and Viresh Rustagi. Experience with a prototype of the POSIX "minimal realtime system profile". In *IEEE Workshop on Real-Time Operating Systems and Software*, pages 12–16, 1994.

[CB90] Deborah Caswell and David Black. Implementing a mach debugger for multithreaded applications. In *Proceedings of the USENIX Conference*, pages 25–40, Winter 1990.

[GB92] E.W. Giering and T.P. Baker. Using POSIX threads to implement Ada tasking: Description of work in progress. In *TRI-Ada '92 Proceedings*, pages 518–529. ACM, November 1992.

[GMB93] E.W. Giering, Frank Mueller, and T.P. Baker. Implementing Ada 9x features using POSIX threads: Design issues. In *TRI-Ada '93 Proceedings*, pages 214–228. ACM, September 1993.

[IEE90] IEEE. *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*, 1990. IEEE Std 1003.1-1990.

[MRB94] F. Mueller, V. Rustagi, and T. P. Baker. Mythos – a micro-kernel threads operating system. In *Symposium on Operating Systems Design and Implementation*, November 1994. (submitted).

[Mue93] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[Rus93] Viresh Rustagi. Implementation and evaluation of a posix threads kernel on a bare machine. Master's thesis, Department of Computer Science, Florida State University, December 1993.

[SPA92] SPARC International, Inc. *The SPARC Architecture Manual: Version 8*, 1992.

[SUN90] SUN Microsystems, Inc. *The SPARCengine 1E Card Family User's Manuals*, April 1990. Part No: 800-8137-02.

[SUN91] SUN Microsystems, Inc. *Open Boot PROM Toolkit User's Guide*, January 1991. Part Number: 800-5279-10.

[Tec92] Technical Committee on Operating Systems and Application Environments of the IEEE Computer Society. *Information Technology - Standardized Applications Environment Profile - POSIX Realtime Application Suport (AEP)*, February 1992. P1003.13/D5.

[Tec94] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C Language] (Draft 9)*, May 1994. P1003.1c/D9.

[Uni] New York University. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.

[U.S83] U.S. Department of Defense. *Military Standard Ada Programming Language*, January 1983. ANSI/MIL-STD-1815A, Ada Joint Program Office.

Vita

Ashwin Goyal was born on January 2, 1971 in Cambay, India. He graduated from R. R. Ruia Junior College, Bombay in 1988. In the fall of 1988 he enrolled as an undergraduate at the Victoria Jubilee Technical Institute, Bombay, India. He graduated in the Spring of 1992 with a Bachelor of Engineering degree in Computer Technology.

In the Fall of 1992 he joined the The Florida State University to pursue a Masters of Science degree in Computer Science. During this time he was a graduate research assistant with the Department of Computer Science . His professional interests includes the areas of real-time systems, compilers and computer architecture.