Multithreading Your Ada Applications

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-4865-11
Revision B, August 1994

© 1994 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, the Sun logo, Sun Microsystems, Sun Microsystems Computer Corporation, Solaris, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark of Novell, Inc., in the United States and other countries; X/Open Company, Ltd., is the exclusive licensor of such trademark. OPEN LOOK® is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

VADS, VADScross, and Verdix are registered trademarks of Rational Software Corporation (formerly Verdix).

The OPEN LOOK and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Please
Recycle

Adobe PostScript

Contents

Figures

...needs a parallelism of life,
a community of thought,
a rivalry of aim

Henry Brooks Adams

Introduction 1

1.1 Two-Tiered Licensing for Solaris Threads

In addition to the previously supported runtime kernel that simulates multithreading, SPARCompiler Ada now supports Solaris threads by means of a second runtime supplied with the product. The threaded runtime allows Ada tasking programs to make full use of Sun's multiprocessor architectures. This threaded capability is called SPARCworks/iMPact Ada, and it is enabled by an additional SunSoft right-to-use license.

In order for you to use SPARCworks/iMPact Ada, you need:

- A fully licensed SPARCompiler Ada installed on your system
- A SPARCworks/iMPact Ada right-to-use license

The SPARCworks/iMPact Ada right-to-use license certificate gives you the name and version number of the licensed SunSoft software product, as well as the maximum number of users who can simultaneously use the product. Your rights are granted and restricted by the Software License Agreement attached to the CD-ROM case. Contact your SunSoft product reseller for information about evaluating or purchasing SPARCworks/iMPact Ada.

After you have read and agreed to the terms of that agreement, follow the instructions in the installation manual appropriate to this product (or other instructions given to you by your software supplier) to request a password which will enable the software. Usually you use your authorization code to transmit and receive a license permission number.

1.2 Contents of this Document

This document describes SPARCompiler Ada support for the product SPARCworks/iMPact Ada, the Solaris MT runtime (using the Solaris Threads microkernel) and ways to configure your application in a multithreaded environment.

Figure 1-1 illustrates the release directory structure for the SPARCompiler Ada self-host product with the directories specific to SPARCworks/iMPact Ada highlighted.

Choose the runtime system you want for your application when you make your application Ada libraries with `a.mklib`. The Solaris MT runtime is indicated by *ada_location* `/self_thr/standard`. This runtime has Solaris Threads as its microkernel.

In this document, Solaris MT refers to the runtime that has Solaris Threads as its kernel. The VADS Threaded runtime has VADS_MICRO as its microkernel. This form of multithreaded Ada is also referred to as MT Ada.

Refer to the material in the Runtime System Guide as you continue with this chapter and manual.

References:

SPARCompiler Ada Runtime System Guide
Refer to the *SunOS Guide to Multithreaded Programming* for additional information on the Solaris multithread architecture.

.

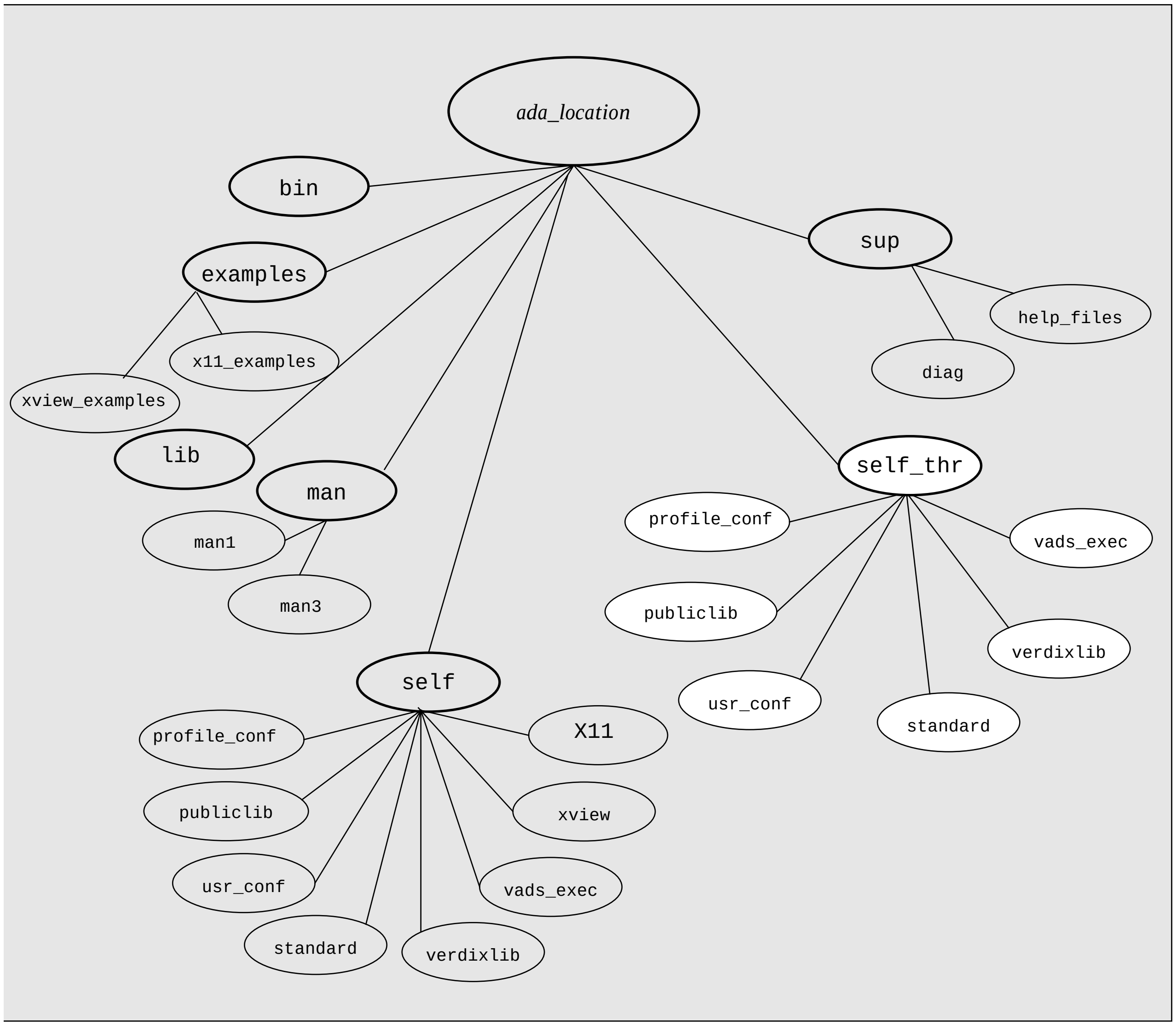

Figure 1-1 SPARCompiler Ada Directory Structure

To use the Solaris MT runtime, you must compile your application with `ada_location /self_thr/standard` on your Ada library's `ADAPATH`.

For example, when using `a.mklib` to create an Ada library to be used with the Solaris MT runtime, you will get output similar to the following:

```
2 versions of SC Ada are available on this machine:
   Target Name   Version     SC Ada Location
 1 SELF_TARGET     2.1       /usr2/ada_2.1/self
                   host_name, host_os, version_number
 2 SELF_TARGET     2.1       /usr2/ada_2.1/self_thr
                   host_name, host_os, version_number
Selection (q to quit):
```

Select 2 to use the Solaris MT runtime.

Caution – Since SPARCworks/iMPact Ada uses a different `standard` library, you cannot have both multithreaded and non-multithreaded libraries on your `ADAPATH.`

"Hold on to threads during sewing of the first few stitches. This eliminates tangling."

Kenmore Ultra-Stitch Owner's Manual

General Threads Overview 2

2.1 The Threaded Runtime System

A threaded runtime system provides support for Ada tasks running in a "threaded" environment. Threaded Ada tasking takes advantage of operating system facilities which implement the parallel execution of "threads" within a single process address space. Often, the operating system's implementation of threads exploits the ability to run threads in parallel on multiple processors, but this is not the only advantage of using threads.

Even on a single processor system, multiple operating system processes can be used effectively to execute the threads defined in a single application program. As described in the sections that follow, a thread-based runtime system increases the degree of parallelism possible within a given Ada application. This increase in parallelism often results in better overall application throughput.

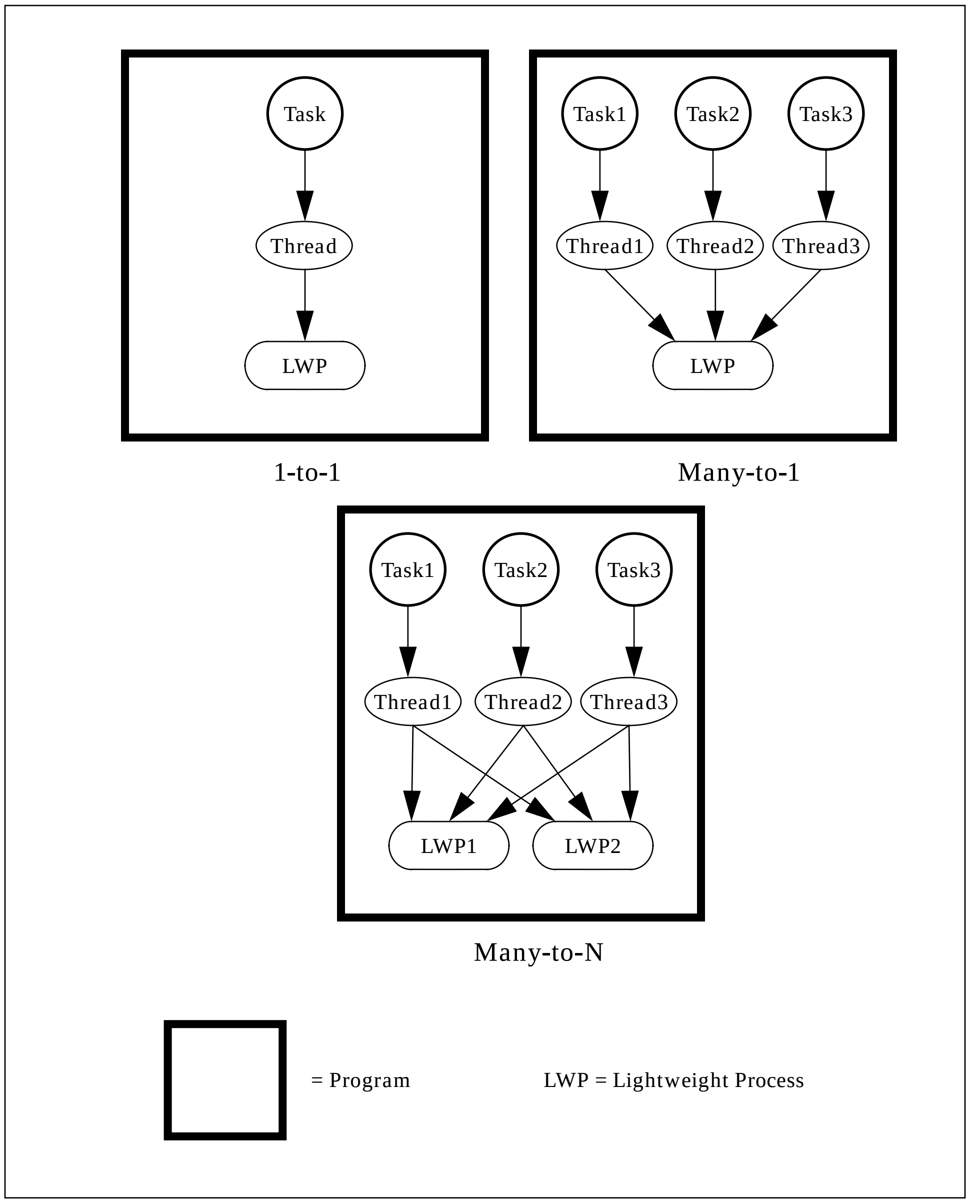

Figure 2-1 Relationship Between Tasks, Threads, and Processes

Every operating system implements threads differently, but all thread implementations share some common characteristics:

1. Threads share most of the resources of an operating system process, including address space, file descriptors and authorization information.
2. Threads achieve parallel operation (on a multiprocessor system) or interleaved execution (on a single processor system) with lower context switching overhead than an equivalent implementation that uses operating system processes. Most of the performance improvement is achieved by eliminating unnecessary calls to the operating system kernel.
3. Although some operating systems support threads directly in the operating system kernel, most thread implementations are layered on top of a “lightweight process” model. A lightweight process is a process that shares the address space and other system resources of its parent process. The parent process can create multiple child lightweight processes, all sharing the same address space, to execute threads.

The Solaris MT Ada runtime system builds Ada tasks on top of the operating system's threads facility. For each Ada task there is a corresponding operating system thread. The Ada runtime system implements the semantics of Ada tasks, including the following: elaboration, rendezvous, abort, and exceptions. The operating system takes care of the low level details of multiplexing one or more threads across a possibly smaller number of lightweight processes. Figure MT - 2 illustrates some of the possible relationships between Ada tasks, threads, and processes. Notice that there is always one thread for every Ada task, and that there may be more threads than lightweight processes.

2.2 Lightweight Processes

Lightweight processes are a special form of the more generic operating system process. A lightweight process shares the address space of it's parent, whereas a normal process has it's own address space, distinct from its parent. When one lightweight process changes a memory location all other processes with the same address space will immediately see the changed value. On Solaris, a lightweight process is called an LWP.

The Solaris MT multithreaded Ada runtime system uses multiple LWPs to execute a single Ada program. The easiest way to understand how this works is to think of the program and its address space as the memory of a computer. Then think of each LWP as a computer processor, executing from this memory.

This model presents a shared memory symmetric multiprocessor, where the LWPs act as symmetric multiprocessors with the program and its address space acting as the shared memory.

The operating system kernel schedules lightweight processes directly. On systems with multiple processors, the kernel may schedule several sibling lightweight processes so that they execute in parallel on different processors. On single processor systems, the kernel executes lightweight processes in an interleaved fashion.

Since lightweight processes are managed by the operating system kernel, they have to make system calls to the kernel when they need to interact with, or synchronize their activities with other lightweight processes. System calls are fairly fast, but if there is a lot of interaction between lightweight processes, the system call overhead may be noticeable.

References

SunOS 5.2 Guide to Multithreaded Programming

2.3 Threads

A thread is an operating system object that manages the flow of control within a single user process but is not directly visible to the operating system kernel. Threads offer well-defined, efficient methods to describe the parallel paths (or threads) of execution within a program.

A thread represents a sequential activity within a program. All the threads within a given program share the main process's address space, file descriptors and other information. Each thread has its own copy of the processor's register set and a subprogram call stack where it keeps local variables, temporary results and return addresses. Collectively, the stack, register set and other information unique to a thread is called its context.

2.4 Multiplexing Threads onto Lightweight Processes

The execution of a Multithreaded Ada program can be modelled as the execution of a shared memory symmetric multiprocessor (SMP) computer. On such a computer, a processor executes a process until the process blocks or is preempted. The processor then saves the context of the process and goes to the ready queue to pick up a new process to execute. Because the system has shared memory and the processors are symmetric, any processor can execute any process.

The Ada application and its memory space are like the memory of an SMP computer and the LWPs are like the processors. To complete the model, threads are comparable to the processes being executed by the SMP processors.

Within a given application there are usually many more threads than underlying lightweight processes that can execute them. This many-to-one relationship is natural because very often many threads are in a suspended state, awaiting the arrival of a particular event. This event might be triggered by the arrival of an asynchronous signal, or by the explicit action of another thread.

When a thread reaches a point where it must suspend its execution, it transfers control to a program known as the thread scheduler. The thread scheduler manages the execution of all threads within a given program. The thread scheduler binds threads onto lightweight processes in an effort to best utilize the available processing resources, and to ensure that higher priority threads are serviced before lower priority threads.

The operating system kernel in turn schedules lightweight processes onto all available processors. The operating system kernel is unaware of the fact that a given lightweight process is being multiplexed between several different threads. The operating system kernel is aware only of system level events such as page faults, disk input/output and process level signals.

The relationship between threads, the operating system kernel and the available system processors is shown in Figure 2-2.

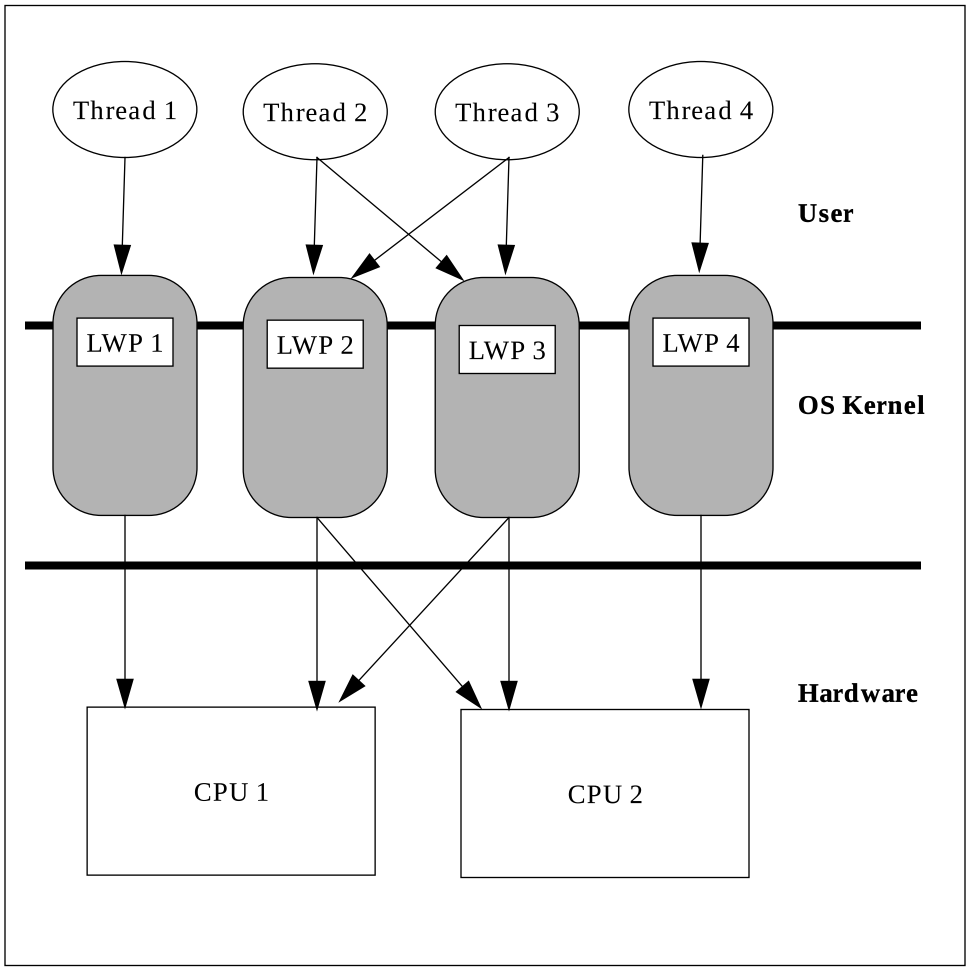

Figure 2-2 Layering of Threads, LWPs, and CPUs

Certain efficiencies result from separating the threads scheduler from the operating system kernel. The threads scheduler is part of the application program and runs in user mode. Therefore, calls can be made to the thread scheduler without invoking a system call. The elimination of system calls makes thread interactions proceed much more quickly than a similar interaction between processes.

2.5 Blocking and Concurrency Level

Often, a thread makes a system call on its own to perform some necessary function. For example, when a thread makes an I/O request, the operating system kernel “blocks” the thread's underlying lightweight process until the I/O operation completes and runs some other eligible process in the meantime.

Once a lightweight process is blocked within the kernel, it is not able to switch to another thread until the system call completes. A thread can keep a lightweight process busy by running for long periods of time without voluntarily giving up control of its lightweight process. It does not matter whether a thread is executing a lengthy calculation, or is blocked inside the operating system kernel waiting for an input/output operation to complete.

If all of the LWPs ina process are blocked in indefinite waits and the process contains a thread that is waiting to run, the threads scheduler creates a new LWP and the appropriate thread is assigned to it for execution.

Another way of eliminating the situation where threads are ready to run, but have no eligible lightweight process to bind to them, is to create additional lightweight processes.

It is not always better to create more lightweight processes than you need. Apart from consuming system resources, lightweight proto threads may not perform thread-level interactions as quickly as they might have if the threads were multiplexed onto a single lightweight process. The reason for this is that if a thread running on one lightweight process needs to signal an event to another thread, then it must execute a system call because the event must be propagated across operating system process boundaries. As we have mentioned before, a system call is much more expensive than a simple procedure call to the thread scheduler.

If there are an exscessive number of LWPs in the pool of a process, and LWPs LWPs in the pool remain idle for a certain period, the threads scheduler destroys the unneeded ones. LWPs are deleted if they are unused for a “long” time, currently five minutes.

As a general rule, you should create enough lightweight processes to run all of your concurrent execution activities plus your I/O activities.

"Many hands make light the work."

John Heywood

Solaris MT and the Threaded Ada Runtime 3

3.1 Overview of the Solaris MT Ada Runtime

This section gives an overview of the Solaris MT Ada runtime and the other operating system facilities that it depends on. The Solaris MT Ada runtime provides tasking services for Ada programs running on Solaris 2.2 or higher computer systems. The Solaris MT Ada tasking services are built on top of the Solaris threads library.

The Solaris threads library is a collection of services that:

- Create threads and synchronization objects (such as mutexes, which control access to "mutual exclusion" regions)
- Establish scheduling attributes (such as thread execution priority)
- Ensure the proper sequencing of task execution, based upon operations on synchronization objects
- Multiplex the execution of a collection of threads onto a (possibly smaller) set of LWP s. These LWP s are in turn scheduled for execution by the operating system to run on one or more processors

A key benefit of the Solaris threads library is that it efficiently multiplexes the execution of a possibly larger collection of threads over a smaller set of LWPs. Since most thread interactions are managed completely outside of the operating system kernel, these interactions are generally an order of magnitude faster than a one-on-one (thread-on-process) implementation. In addition, since a thread's “context” is much smaller than that of an operating system process (even a lightweight process), it consumes less memory.

Threads offer an efficient primitive for managing concurrency; they make an ideal starting place for the implementation of an Ada runtime system. While the threads library focuses on scheduling threads for execution on one or more processors, the Ada runtime need only concern itself with implementing Ada tasking semantics. This layering is shown in Figure 3-1.

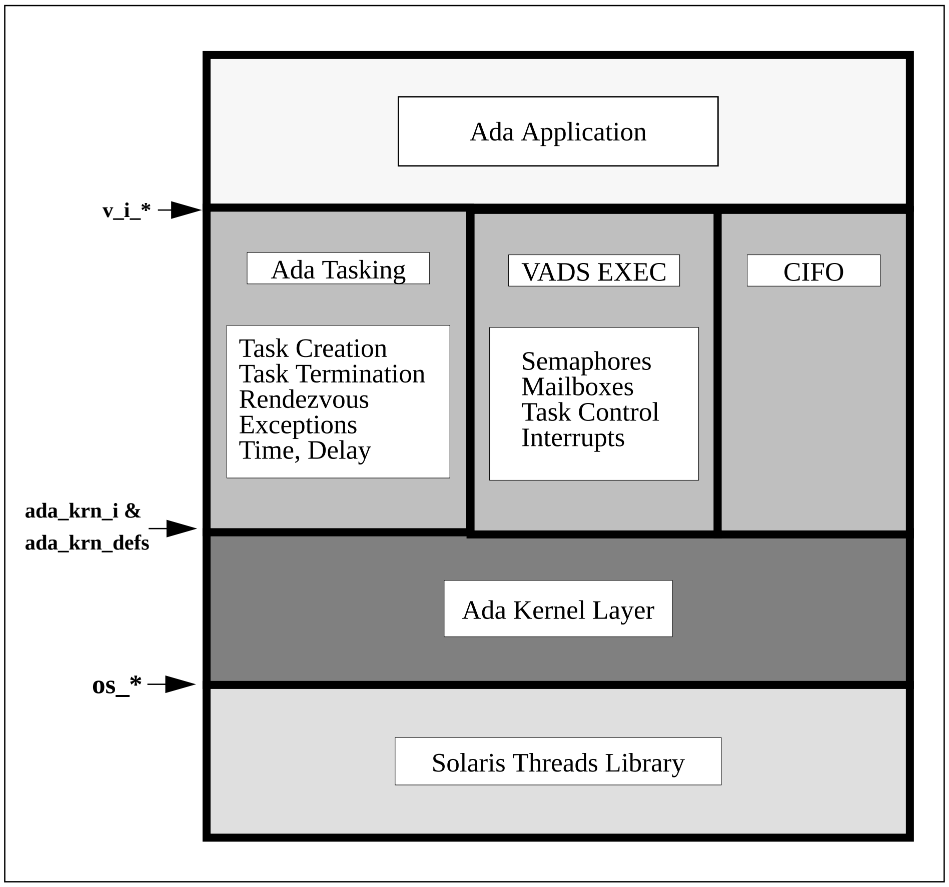

Figure 3-1 Threaded Ada Runtime Layers

The Solaris MT Ada runtime system implements the semantics of Ada tasking in a set of procedures that are called implicitly from code generated by the Ada compiler. These procedures are defined in detail in package `V_I_TASKOPS` in the *ada_location*`/self_thr/standard` directory. These tasking procedures will in turn make one or more calls to a layer known as the “Ada Kernel”. The Ada Kernel's procedures are defined in the package, `ada_krn_i.a` and the datatypes associated with these procedures are defined in the package, `ada_krn_defs.a.` Both of these Ada kernel interface definition packages are located in the `standard` Ada library.

The Ada kernel offers various synchronization and control operations that are customized to fit well with the needs of the Ada runtime system.

The Solaris MT Ada kernel makes calls to the Solaris threads library, which implements operations on threads, mutexes, semaphores, and condition variables.

Five Ada binding packages define the interface to the Solaris threads library:

- OS_ALLOC implements memory allocation
- OS_SIGNAL has the signal contents and services
- OS_SYNCH implements operations on mutexes, condition variables and semaphores
- OS_THREADS implements operations on Solaris threads
- OS_TIME implements time services.

The Ada runtime system focuses on the details of implementing Ada tasking operations such as task activation, rendezvous, and abort. The Ada kernel layer implements specific lower level services required by the Ada runtime. The Ada kernel calls the operating system to create and control threads and mutexes, and to interact with the operating system defined method of handling traps and interrupts.

3.2 Task Attributes

Most of the time you do not need to be concerned with the layering of Ada tasks onto operating system threads. However, when you require specific scheduling behavior, you need a method to pass information from the Ada program to the operating system's threads library. The object that passes thread-related scheduling information for an associated Ada task is known as the “task attribute” structure.

Each Ada task has a “task attribute” record that describes the attributes of a task which are established when the task is first created and activated. The task attribute record is defined in the package `ada_krn_defs.a`, located in the standard Ada library.

```
 --------------------------------------------------------------
-
 -- Task attribute types
 --------------------------------------------------------------
-
This record type is used for passing OS specific task
-- information at task create.
--
-- Note: the priority in the task_attr_t record takes precedence
-- over that specified by "pragma priority()".
--
-- The prio and cond_attr_address fields are referenced by
-- the Ada rts.
--
-- The mutex_attr_address is the address of a mutex_attr_t
record.
-- The cond_attr_address is the address of a cond_attr_t record.
-- Setting these fields to NO_ADDR selects the default values
-- specified by the DEFAULT_TASK_ATTRIBUTES parameter in
-- v_usr_conf's configuration_table.
   flags:           integer;
end record;
```

Figure 3-2 Task Attribute Types

Each of the fields in the task attribute record plays an important role in the execution of an Ada task and the operating system thread associated with it. Each field is introduced below:

- The `prio` field determines the scheduling priority used when the task becomes ready to run. Eligible tasks are chosen on a highest-priority-first basis.

- The flags field specifies the value of the flags parameter passed to the `thr_create()` service when the task is created. The Ada RTS always sets the `THR_SUSPENDED` and `THR_DETACHED` attributes. You can optionally set the `THR_BOUND` and / or `THR_NEW_LWP` attributes.
- The `mutex_attr_address` field contains the address of the attributes to be used to initialize the mutex object implicitly created for the task. This mutex is used to protect the task's data structure. For example, the task's mutex is locked when another task attempts to rendezvous with it. If you set `mutex_attr_address` to `NO_ADDR`, the `mutex_attr_address` value specified by the `V_USR_CONF.CONFIGURATION_TABLE` parameter, `DEFAULT_TASK_ATTRIBUTES`, is used. Otherwise, `mutex_attr_address` must ber set to the address of an `ADA_KRN_DEFS.MUTEX_ATTR_T` record. Initialize the `MUTEX_ATTR_T` record using one of the `ADA_KRN_DEFS` mutex attribute init subprograms.
- The `cond_attr_address` field contains the address of the attributes to be used to initialize the condition variable object implicitly created for the task. When the task blocks, it waits on this condition variable. If you set `cond_attr_address` is set to `NO_ADDR`, the cond_attr_address value specified by the `V_USR_CONF.CONFIGURATION_TABLE` parameter, `DEFAULT_TASK_ATTRIBUTES` is used. Otherwise, `cond_attr_address` must be set to the address of a `ADA_KRN_DEFS.COND_ATTR_T` record. The `COND_ATTR_T` record should be initialized using one of the `ADA_KRN_DEFS` condition variable attribute init routines.

References

Condition Variable Support Subprograms, *SPARCompiler Ada Runtime System Guide*

Most of the time you will use the default task attributes. However, in certain situations, you may need to directly control the scheduling behavior of your program's Ada tasks. To understand the effect of setting a task's scheduling attributes, you need to know how the threads scheduler binds threads to lightweight processes and how lightweight processes are in turn scheduled by the operating system.

3.3 Task Priority and Process Priority

An Ada task's priority affects the service that it receives from the group of LWPs that are eligible to run the task's underlying thread.

Since LWPs are managed by the operating system kernel, they compete for processor cycles based upon the LWP‘s system defined priority. If you do not take any specific actions in your program, a LWP‘s priority varies over time; its priority is adjusted by the operating system based upon the amount of CPU time it has accumulated. Most operating systems lower, or “degrade” the priority of a CPU bound LWP and raise the priority of an LWP that needs only short bursts of CPU time. Generally, this default, automatic adjustment of LWP priority performs well and your program does not need to make explicit priority adjustments.

3.4 pragma TASK_ATTRIBUTES

A special Ada pragma, TASK_ATTRIBUTES, is used to associate a given task attribute setting to an Ada task when that task is created.

```
pragma task_attributes(task_attr'address);
pragma task_attributes(task_object, task_attr'address);
```

The first form of the pragma implies the name of the task object declaration that contains pragma TASK_ATTRIBUTES. The second form of pragma TASK_ATTRIBUTES is used to select a particular task object.

pragma TASK_ATTRIBUTES applies the specified attributes to the creation of the affected task. Usually, the task attribute structure has constant values set at compile time. The task attribute structure may be a compile-time constant or it can be changed dynamically when the program runs.

3.5 Default and Main Task Attributes

The default task attribute setting is used any time a task is created and there is no explicit task attribute setting supplied with the pragma TASK_ATTRIBUTES directive. You can change the default task attribute setting by modifying the value of the DEFAULT_TASK_ATTRIBUTES field in the user configuration table. Typically, values are changed in the user configuration table by copying *ada_location*/`self_thr/usr_conf/v_usr_conf_b.a` to an application specific directory, then modifying the source code of `v_usr_conf_b.a` and recompiling.

The `MAIN_TASK_ATTR_ADDRESS` parameter in the user configuration table points to the task attributes for the main program. Since the main program is a library level procedure, the pragma `TASK_ATTRIBUTES` directive cannot be applied, so we get its task attributes from the configuration table.

References

`DEFAULT_TASK_ATTRIBUTES`, *SPARCompiler Ada Programmer's Guide*
`MAIN_TASK_ATTR_ADDRESS`, *SPARCompiler Ada Programmer's Guide*
User Library Configuration, *SPARCompiler Ada Programmer's Guide*

3.6 Creating and Controlling LWPs

The number of LWPs created by your Ada application program and their scheduling characteristics, directly affect the level of concurrency achieved, and in some situations, the program's correctness. Here are a few of the situations in which creating additional LWPs or controlling their scheduling characteristics is desirable:

- You may decide to create more LWPs because some tasks make system calls that suspend or block their execution for long periods of time. When a task blocks by making a system call, the thread scheduler is unable to make the LWP bound to the thread switch to another ready-to-run thread. This ties up the LWP for indefinite periods of time and can result in a situation where there are several tasks that are ready-to-run, but no LWPs to run them.
- You may need to create LWPs dedicated to running a specific set of services. For example, XView services are inherently single threaded and must be run within the context of a single process. In addition, most relational database implementations have not been written to operate properly when several threads of control within a single process access the database.
- On a multiprocessor system, you may need to explicitly control the allocation of LWPs across the available physical processors. You need to override the operating system's automatic load balancing procedure when you want to tightly control a task's response time to a given external event, or when there is a hardware device that can be accessed only from a specific processor.

3.7 Setting the Concurrency Level

By default, the Ada runtime system does not create additional LWPs to run your application's Ada tasks. All threads are run on only the initial main process.

You can change this default behavior in one of the following ways:

1. Copy ada_location `/self_thr/usr_conf/v_usr_conf_b.a` into your Ada library (or a user parent library) and change the `CONCURRENCY_LEVEL` in the configuration table from its default value of 1 to the desired value.
2. Leave the value of `CONCURRENCY_LEVEL` set to 0 or 1 and set the environment variable, `ADA_CONCURRENCY_LEVEL` to the desired number of LWPs. Since the environment variable can be changed without recompiling and linking the application, this method of setting the LWP count makes it easy to experiment during the development cycle.
3. Call the Solaris threads service, `thr_setconcurrency()`from your Ada application.

References

`CONCURRENCY_LEVEL`, *SPARCompiler Ada Programmer's Guide*

3.8 Dedicating a LWP to a Specific Task

In certain situations, it may be undesirable to let a task "float" from one LWP to another. You may want a LWP to be dedicated to run a specific task and only that task. To ensure that a LWP is dedicated to a particular task, you must set the thread attribute, `thr_bound`, in the task attribute's flags field.

3.9 Signals

This chapter describes signal handling in Solaris MT Ada. First it provides an overview of signal handling in Solaris threads. Then it shows how Solaris MT Ada layers on Solaris threads to provide signal handling in an application program. Differences from VADS MICRO signals are highlighted. The chapter concludes by talking about the configuration parameters added to `v_usr_conf` to support signals in Solaris MT Ada.

3.9.1 Overview of Signals in Solaris Threads

The following is an extraction from *SunOS 5.2 Guide to Multi-Thread Programming, Chapter 5, Signals.*

3.9.1.1 Signal Handling

Each thread has its own signal mask. This lets a thread block some signals while it uses memory or other state that is also used by a signal handler. All threads in a process share the set of signal handlers set up by `signal(2)` and its variants, as usual.

If a signal handler is marked `SIG_DFL` or `SIG_IGN`, the action on receipt of the signal (`exit`, `core dump`, `stop`, `continue`, or `ignore`) is performed on the entire receiving process and affects all the threads in the process.

Signals are divided into two categories: traps and interrupts.

Traps (such as `SIGILL`, `SIGFPE`, `SIGSEGV`) result from execution of a specific thread and are handled only by the thread that caused them. Several threads in a process could generate and handle the same type of trap simultaneously.

Interrupts (such as `SIGINT`, `SIGIO`) are asynchronous with any thread and result from some action outside of the process. An interrupt can be handled by any thread whose signal mask has it enabled. If more than one thread is able to receive the interrupt, only one is chosen. One result is that several threads can be in the process of handling the same kind of signal simultaneously. If all threads mask a signal, it will remain pending on the process until some thread enables the signal. As in single-threaded processes, the number of signals received by the process is less than or equal to the number sent. For example, an application can enable several threads to handle a particular I/O interrupt. As each new interrupt comes in, another thread is chosen to handle the signal until all the enabled threads are active. Any additional signals remain pending until a thread completes processing and re-enables the signal.

Threads can send signals to other threads in the process through `thr_kill(3T)`. A signal initiated through `thr_kill(0)` behaves like a trap and is handled only by the specified thread. As usual, a signal is sent to another process by `kill(2)` or `sigsend(2)`. There is no direct way for a thread in one process to send a signal to a specific thread in another process.

3.9.1.2 Async-safe Functions

Functions that may be called from signal handlers are said to be async-safe. In the threads library the following functions are async-safe in addition to those defined by *POSIX (IEEE std 1003.1-1990, 3.3.1.3 (3)(f),* page 55):

- `sema_post()`
- `thr_sigsetmask()`
- `thr_kill()`

3.9.1.3 Thread Signal Interface

Each thread has its own signal mask. A thread's signal mask applies equally to intra-process signals (signals sent between threads) and to enterprises signals.

thr_sigsetmask(3T)

Sets the thread's signal mask. A thread's initial signal mask is inherited from the parent thread.

thr_kill(3T)

Causes the specified signal to be sent to a specified thread.

3.9.1.4 Waiting for Signals

A new way to handle asynchronous signals is to wait for them in a separate thread. This is safer and easier than installing a signal handler and processing them there.

sigwait(2)

Wait for a pending signal from the set specified by the argument, regardless of the signal mask. `sigwait()` clears the pending signal and returns its number. If more than one thread is waiting for the same signal, then one thread is chosen and it returns from `sigwait()`.

`sigwait()` is typically used by creating one or more threads that wait for signals. Since `sigwait()` can retrieve even masked signals, the signals of interest are usually blocked in all threads so they are not accidentally delivered. When the signals arrive a thread returns from `sigwait()`, handles the signal and waits for more signals. The signal handling thread is not restricted to using `async-safe` functions and can synchronize with other threads in the usual way.

3.9.2 Signals in Solaris MT Ada

The Ada RTS installs handlers for the `SIGILL`, `SIGFPE`, and `SIGSEGV` synchronous signals. These handlers map the signals into Ada exceptions.

The Ada RTS uses the `sigwait()` service for handling all other signals.

When the application attaches a signal handler either explicitly via `V_XTASKING.ATTACH_ISR` or implicitly via an interrupt entry, an interrupt task is created for the attached signal handler. The interrupt task loops doing a `sigwait()` for the attached signal. When the process gets the asynchronous signal, `sigwait()` returns and the interrupt task calls the ISR procedure. Since the ISR procedure is executing in the context of an Ada task, there are no restrictions on the Ada operations it can perform or the VADS EXEC services it can call. This differs radically from the VADS MICRO where the ISR procedure executes in the context of the UNIX signal handler and is very limited in what it can do or call.

Use the `lt` command in the debugger to display the interrupt tasks. Here is the debugger lt output when the application has attached handlers for the `SIGINT`, `SIGUSR1`, and `SIGUSR2` signals:

```
>lt
 Q  TASK             ADDR       STATUS
    <interrupt 17>  00009981c  waiting for interrupt
         handler at 000034ad0
    <interrupt 16>  0000996ec  waiting for interrupt
         handler at 000034ab0
    <interrupt 2>   0000995bc  waiting for interrupt
         handler at 000034a90
 *  <main program>  00005866c  executing
```

Figure 3-3 Display Interrupt Tasks With `lt`

The handler at value is the address of the ISR procedure called in the interrupt task upon return from `sigwait()`.

A necessary requirement for using `sigwait()` reliably is to ensure that the signal it waits on is disabled in all threads. To satisfy this need, `thr_sigsetmask()` is called during program initialization to disable all asynchronous signals. This signal mask is inherited by all subsequently created threads.

The VADS EXEC services, `V_INTERRUPTS.CURRENT_INTERRUPT_STATUS` and `V_INTERRUPTS.SET_INTERRUPT_STATUS` call the thread service, `thr_sigsetmask()` to get or set the signal mask for the current thread. This differs from the VADS_MICRO where there is one signal mask applicable to all the tasks. In the VADS MICRO, `V_INTERRUPTS.SET_INTERRUPT_STATUS` is called to disable signals for all tasks. Solaris threads does not have a service to atomically change the signal mask for all threads.

Given the above requirement for `sigwait()`, asynchronous signals should always remain disabled. The application should avoid calling `V_INTERRUPTS.SET_INTERRUPT_STATUS` to enable any additional signals.

Since ISRs execute in the context of a task and since signals can be disabled / enabled only on a per task basis, the application cannot use `V_INTERRUPTS.SET_INTERRUPT_STATUS` to protect critical regions using Solaris threads as can be done using the VADS MICRO. However, since the ISR executes in the context of a task it can block waiting on a rendezvous, passive task entry, mutex or semaphore.

3.9.3 Configuration Table Parameters for Solaris MT Ada Signals

Parameters have been added to the configuration table in `v_usr_conf_b.a` to support signal handling in Solaris MT Ada. These parameters specify which signals are always disabled, which signals when not attached cause the program to exit, and the priority and stack size of all interrupt tasks.

An overview of the Solaris MT Ada signal configuration parameters follows:

```
ENABLE_SIGNALS_MASK
ENABLE_SIGNALS_33_64_MASK
```

These masks are used during startup to initialize the main task's signal mask. All subsequently created threads / tasks inherit this signal mask.

A separate Ada interrupt task is created for each attached signal handler. This task does a `sigwait()` for the attached signal. Therefore, a handler may be attached only for a signal disabled by these enable masks.

By default `ENABLE_SIGNALS_MASK = 16#FEBF_B007#` and `ENABLE_SIGNALS_33_64_MASK = 0`. The default disables all signals except:

4 - `SIGILL` (mapped to an Ada exception)
5 - `SIGTRAP`
6 - `SIGIOT. SIGABRT`
7 - `SIGEM`
8 - `SIGFPE` (mapped to an Ada exception)
9 - `SIGKILL`
10 - `SIGBUS`
11 - `SIGSEGV` (mapped to an Ada exception)
12 - `SIGSYS`
15 - `SIGTERM` (used by Ada RTS for thread termination)
23 - `SIGSTOP`
25 - `SIGCONT`
33 - `SIGLWP` (used by Solaris threads)

Effectively, only the synchronous, exception related signals are enabled.

Since we use `sigwait` to handle asynchronous signals, an asynchronous signal must never be enabled in any thread.

EXIT_SIGNALS_MASK EXIT_SIGNALS_33_64_MASK

Since we normally disable all asynchronous signals, we have another set of signal masks that indicate which signals should cause the program to exit if a handler is not attached. An exit signals thread is created that waits for one of these signals. However, attached signals are automatically removed from the exit signal mask.

Exit signals not included in the enable signals masks are ignored.

By default `EXIT_SIGNALS_MASK = 16#0001_8006#` and `EXIT_SIGNALS_33_64_MASK = 0`. The default causes the program to exit if it receives one of the following signals without an attached handler:

2 - SIGINT
3 - SIGQUIT
16 - SIGUSR1
17 - SIGUSR2

INTR_TASK_PRIO

INTR_TASK_PRIO is the priority of all the interrupt tasks doing a sigwait() for an attached signal. The default is PRIORITY'LAST.

INTR_TASK_STACK_SIZE

INTR_TASK_STACK_SIZE is the size of the stack for all interrupt tasks. The default is the same as DEFAULT_TASK_STACK_SIZE.

3.10 Ada Kernel: Solaris MT Implementation

The Solaris MT implementation of the Ada Kernel differs from the VADS MICRO version. The attributes of the Ada Kernel objects are different. Some of the Ada Kernel services are not supported or have slightly different semantics.

The write-up on the Ada Kernel in Chapter 2 of the *SPARCompiler Ada Runtime System Guide* walks through all the objects in the Ada Kernel. The following topics are addressed for each object: types, constants, attributes, services, and support subprograms. The information provided assumes the Ada Kernel is layered upon the VADS MICRO.

We would like to revisit each of the Ada Kernel objects. This time the discussion will focus on the Solaris MT implementation. Only differences from the VADS MICRO version will be provided.

3.10.1 Ada Program

There is no support for multiple programs. The following services are not supported:

PROGRAM_START - always returns NO_PROGRAM_ID
PROGRAM_GET_KEY - always returns NO_ADDR

3.10.2 Ada Task

The Solaris thread priority for an Ada task is the Ada priority plus one. Since the Ada Kernel has an idle thread with a Solaris priority of zero, the Solaris priority must be biased by one.

KRN_TASK_ID - this is the Solaris thread ID.

The TASK_ATTR_T record contains the flags field in addition to the prio, mutex_attr_address and cond_attr_address fields. The flags field specifies the value of the flags parameter passed to the Solaris thr_create() service when the task is created. The Ada RTS always sets the THR_SUSPENDED and THR_DETACHED attributes. You can optionally set the THR_BOUND and/or THR_NEW_LWP attributes.

The task_attr_init() subprograms have an optional parameter for initializing the flags fields. If omitted, the flags field defaults to THR_BOUND and THR_NEW_LWP not being set.

The following task management services (VADS EXEC augmentation) are supported differently from the VADS MICRO version:

```
TASK_DISABLE_PREEMPTION
TASK_ENABLE_PREEMPTION
```

Disabling preemption is not meaningful in a multiple CPU environment such as Solaris threads. Nevertheless, the preemption services are implemented as follows. A preemption depth count is maintained for each Ada task. It is initialized to zero. TASK_DISABLE_PREEMPTION increments the depth. When it is incremented from 0 to 1, the current task's priority is saved and the priority is elevated higher than any Ada priority. TASK_ENABLE_PREEMPTION decrements the depth. When decremented to 0, the task priority saved when the preemption was initially disabled, is restored.

```
TASK_SUSPEND
TASK_RESUME
```

These services map directly to the Solaris thread services, thr_suspend() or thr_continue().

The following task management services (VADS EXEC augmentation) are not supported:

`TASK_GET_TIME_SLICE` - always returns 0.0
`TASK_SET_TIME_SLICE` - silently ignored
`TASK_GET_SUPERVISOR_STATE` - always returns `FALSE`
`TASK_SET_SUPERVISOR_STATE` - silently ignored
`TASK_ENTER_SUPERVISOR_STATE` - silently ignored
`TASK_LEAVE_SUPERVISOR_STATE` - silently ignored

The following sporadic task services (CIFO augmentation) are not supported:

`TASK_IS_SPORADIC` - always returns `FALSE`
`TASK_SET_FORCE_HIGH_PRIORITY` - silently ignored

3.10.2.1 Ada Task Master

No differences.

3.10.2.2 Ada "new" Allocation

The allocation services map directly to the `malloc()` and `free()` routines in the MT-safe C library.

3.10.2.3 Kernel Scheduling

There is no support for controlling kernel scheduling policies. The following services are not supported:

`KERNEL_GET_TIME_SLICING_EANBLED` - always returns `FALSE`
`KERNEL_SET_TIME_SLICING_EANBLED` - silently ignored

3.10.2.4 Callout

`CALLOUT_EVENT_T` only has two program events: `EXIT_EVENT` and `UNEXPECTED_EXIT_EVENT`. The VADS MICRO idle and task events are not supported.

3.10.2.5 Task Storage

There is no support for user-defined storage on a per task basis. The following services are not supported:

```
TASK_STORAGE_ALLOC  - always returns NO_TASK_STORAGE_ID
TASK_STORAGE_GET    - always returns NO_ADDR
TASK_STORAGE_GET2   - always returns NO_ADDR
```

3.10.2.6 Interrupts

ENABLE_INTR_STATUS

Since `sigwait()` is used to wait for and handle all asynchronous signals, all asynchronous signals are always disabled. `ENABLE_INTR_STATUS` masks the same signals as `DISABLE_INTR_STATUS`.

The interrupt services are supported differently from the VADS MICRO version as follows:

INTERRUPTS_GET_STATUS

`INTERRUPTS_GET_STATUS` calls the Solaris routine, `thr_sigsetmask()`, to get the signal mask for the current thread. Signal masks are maintained on a per thread basis and not a per process basis.

INTERRUPTS_SET_STATUS

`INTERRUPTS_SET_STATUS` calls the Solaris routine, `thr_sigsetmask()`, to set the signal mask for the current thread. `INTERRUPTS_SET_STATUS` must not be called to enable any signals with handlers attached via `ISR_ATTACH`.

ISR_ATTACH

`ISR_ATTACH` creates an interrupt task that does a `sigwait()` for the interrupt vector. The ISR procedure is called in the context of an Ada task.

3.10.2.7 ISR_DETACH

If we subsequently receive the detached signal, it is ignored unless it is set in EXIT_SIGNALS_MASK or EXIT_SIGNALS_33_64_MASK in v_usr_conf's configuration table. When not ignored, the program exits.

ISR_IN_CHECK

Since ISR handlers execute in the context of an Ada task, ISR_IN_CHECK always returns FALSE.

3.10.2.8 *Time*

No differences.

3.10.2.9 *Mutex*

The MUTEX_ATTR_T record consists of two fields: typ and arg. These fields correspond to the arguments: type and arg passed to the Solaris threads routine, mutex_init(). The function, DEFAULT_MUTEX_ATTR, sets typ to USYNC_THREAD and arg to NO_ADDR. Since attached signal handlers (ISRs) execute in the context of an Ada task, the function, DEFAULT_INTR_ATTR, does the same as DEFAULT_MUTEX_ATTR.

Since ISRs execute in the context of an Ada task, the following ISR mutex services are supported differently from the VADS MICRO version as follows:

ISR_MUTEX_LOCKABLE	- always returns TRUE
ISR_MUTEX_LOCK	- maps directly onto MUTEX_LOCK
ISR_MUTEX_UNLOCK	- maps directly onto MUTEX_UNLOCK

The following priority ceiling mutex services (CIFO augmentation) are not supported:

CEILING_MUTEX_INIT	- always returns FALSE
CEILING_MUTEX_SET_PRIORITY	- always returns FALSE
CEILING_MUTEX_GET_PRIORITY	- always returns -1

The mutex support subprograms behave differently from VADS MICRO as follows:

FIFO_MUTEX_ATTR_INIT

Solaris threads supports only priority queuing. These init subprograms also select priority queuing and do the same as the PRIO_MUTEX_ATTR_INIT subprograms.

PRIO_MUTEX_ATTR_INIT

Since Solaris threads supports only priority queueing this is the same as the DEFAULT_MUTEX_ATTR. As for the DEFAULT_MUTEX_ATTR, **typ** is set to USYNC_THREAD and arg to NO_ADDR.

Solaris MT Ada does not support priority inheritance. These init subprograms raise the PROGRAM_ERROR exception.

PRIO_MUTEX_ATTR_INIT

Since Solaris MT Ada only supports priority queueing this is the same as the DEFAULT_MUTEX_ATTR. As for the DEFAULT_MUTEX_ATTR, typ is set to USYNC_THREAD and arg to NO_ADDR.

PRIO_INHERIT_MUTEX_ATTR_INIT

Solaris threads does not support priority ceiling mutexes. These init subprograms raise the PROGRAM_ERROR exception.

INTR_ATTR_INIT

Since ISRs execute in the context of an Ada task, the DISABLE_STATUS argument is ignored and the mutex is initialized using the corresponding PRIO_MUTEX_ATTR_INIT subprogram.

3.10.2.10 Condition Variable

The COND_ATTR_T record consists of two fields: typ and arg. These fields correspond to the arguments: type and arg passed to the Solaris threads routine, cond_init(). The function, DEFAULT_COND_ATTR, sets typ to USYNC_THREAD and arg to NO_ADDR.

The COND_WAIT subprogram behaves differently. On Solaris threads a condition variable may be prematurely signaled by an OS stimulus such as delivery of a signal or a fork. Check the condition(3T) man page for more details.

Since ISRs execute in the context of an Ada task, ISR_COND_SIGNAL maps directly to COND_SIGNAL.

The condition variable support subprograms behave differently from VADS MICRO as follows:

FIFO_COND_ATTR_INIT

Solaris threads supports only priority queuing. These init subprograms also select priority queuing and do the same as the PRIO_COND_ATTR_INIT subprograms.

PRIO_COND_ATTR_INIT

Since Solaris threads supports only priority queueing this is the same as the DEFAULT_COND_ATTR. As for the DEFAULT_COND_ATTR, typ is set to USYNC_THREAD and arg to NO_ADDR.

3.10.2.11 Binary Semaphore

Binary semaphores have are implemented using the Solaris threads counting semaphore services.

SEMAPHORE_FULL = 1 and SEMAPHORE_EMPTY = 0.

The SEMAPHORE_ATTR_T record consists of two fields: typ and arg. These fields correspond to the arguments: type and arg passed to the Solaris threads routine, sema_init(). The function, DEFAULT_SEMAPHORE_ATTR, sets typ to USYNC_THREAD and arg to NO_ADDR.

The following binary semaphore services are supported differently from the VADS MICRO:

SEMAPHORE_TIMED_WAIT

Solaris threads does not have a timed wait service for semaphores. The timed wait is emulated by doing a SEMAPHORE_TRYWAIT followed by a TIME_DELAY. This trywait/delay is repeated until the trywait is successful or it has timed out. The TIME_DELAY is passed the following delay times: 0.1, 0.5, 1.0, 2.0, 5.0 and 10.0. Once the delay time of 10.0 is used, it's used for all subsequent delays.

SEMAPHORE_GET_IN_USE

Since Solaris threads does not have a service to check if any task is waiting on a semaphore, SEMAPHORE_GET_IN_USE always returns TRUE.

3.10.2.12 *Counting Semaphore*

The COUNT_SEMAPHORE_ATTR_T record consists of two fields: typ and arg. These fields correspond to the arguments: type and arg passed to the Solaris threads routine, sema_init().

The function, DEFAULT_COUNT_SEMAPHORE_ATTR, sets typ to USYNC_THREAD and arg to NO_ADDR. Since the Ada Kernel's counting semaphores are layered directly on the counting semaphores provided in Solaris threads, the DEFAULT_COUNT_INTR_ATTR function and the COUNT_INTR_ATTR_INIT subprograms do the same as DEFAULT_COUNT_SEMAPHORE_ATTR. The DISABLE_STATUS parameter is ignored in the COUNT_INTR_ATTR_INIT subprograms.

Note – A counting semaphore in Solaris threads can be signaled from a Solaris signal handler.

The following counting semaphore services are supported differently from the VADS MICRO:

COUNT_SEMAPHORE_WAIT

Solaris threads does not have a timed wait service for semaphores. The timed wait option (wait_time > 0.0) is emulated by doing a COUNT_SEMAPHORE_TRYWAIT followed by a TIME_DELAY. This trywait/delay is repeated until the trywait is successful or it has timed out. The TIME_DELAY is passed the following delay times: 0.1, 0.5, 1.0, 2.0, 5.0 and 10.0. Once the delay time of 10.0 is used, its used for all subsequent delays.

COUNT_SEMAPHORE_GET_IN_USE

Since Solaris threads does not have a service to check if any task is waiting on a semaphore, COUNT_SEMAPHORE_GET_IN_USE always returns TRUE.

3.10.2.13 Mailbox

Solaris threads has no direct support for mailboxes. Consequently, mailboxes are implemented in the Ada Kernel using a mutex to protect the mailbox data structures and using a condition variable when waiting to read a message.

MAILBOX_ATTR_T is a subtype of MUTEX_ATTR_T. The functions DEFAULT_MAILBOX_ATTR and DEFAULT_MAILBOX_INTR_ATTR map directly to the default attribute initialization functions for mutexes. The subprograms MAILBOX_INTR_ATTR_INIT map directly to the mutex INTR_ATTR_INIT subprograms.

Since Solaris threads does not have a service to check if any task is waiting on a condition variable, MAILBOX_GET_IN_USE always returns TRUE.

3.11 VADS EXEC: Solaris MT Ada Differences

The VADS EXEC services are layered upon the Ada Kernel. In the Solaris MT implementation of the Ada Kernel, some of the services are not supported or have slightly different semantics from the VADS MICRO version. Therefore, how the Ada Kernel is implemented using Solaris threads has a direct impact on the behavior of the VADS EXEC services in MT Ada.

All of the VADS EXEC services are documented in Chapter 4 of the Runtime System Guide. The services are partitioned across the following packages: V_INTERRUPTS, V_MAILBOXES, V_MEMORY, V_SEMAPHORES, V_XTASKING, and V_STACK. The write-up assumes the underlying micro kernel is the VADS MICRO.

We would like to revisit each of the services in the VADS EXEC packages. This time we will assume that Solaris threads is the underlying micro kernel. Only differences from the VADS MICRO version will be provided.

3.11.0.1 V_INTERRUPTS

The following interrupt services are supported differently from the VADS MICRO version:

ATTACH_ISR

ATTACH_ISR creates an interrupt task that does a sigwait() for the interrupt vector. Since the ISR procedure is called in the context of an Ada task, there are no restrictions on the services that can be called.

CURRENT_INTERRUPT_STATUS

CURRENT_INTERRUPT_STATUS calls the Solaris routine, thr_sigsetmask(), to get the signal mask for the current thread. Signal masks are maintained on a per thread basis and not a per process basis.

For DETACH_ISR, if we subsequently receive the detached signal, it is ignored unless it is set in EXIT_SIGNALS_MASK or EXIT_SIGNALS_33_64_MASK in v_usr_conf's configuration table. When not ignored, the program exits.

SET_INTERRUPT_STATUS

SET_INTERRUPT_STATUS calls the Solaris routine, thr_sigsetmask(), to set the signal mask for the current thread. SET_INTERRUPT_STATUS must not be called to enable any signals with handlers attached via ATTACH_ISR.

The following interrupt services are not supported:

CURRENT_SUPERVISOR_STATE	- always returns FALSE
ENTER_SUPERVISOR_STATE	- silently ignored
LEAVE_SUPERVISOR_STATE	- silently ignored
SET_SUPERVISOR_STATE	- silently ignored

3.11.0.2 V_MAILBOXES

The services in V_MAILBOXES are layered upon the mailbox services provided in the Ada Kernel. Refer to the previous section on the Solaris MT implementation of mailboxes in the Ada Kernel.

Since ISRs execute in the context of an Ada task, an ISR can access the mailbox without disabling interrupts.

For DELETE_MAILBOX, since its unable to detect if any task is waiting to read from the mailbox, it always assumes the mailbox is in use. Therefore, if DELETE_MAILBOX is called with DELETE_OPTION set to DELETE_OPTION_WARNING, it will never be deleted. On the other hand, if DELETE_MAILBOX is called with DELETE_OPTION set to DELETE_OPTION_FORCE, it does a dummy mailbox write and waits 3.0 seconds before freeing the mailbox resources.

3.11.0.3 V_MEMORY

No differences.

3.11.0.4 V_SEMAPHORES

The services in V_SEMAPHORES are layered upon the binary or counting semaphore services provided in the Ada Kernel. Refer to the previous section on the Solaris MT implementation of semaphores in the Ada Kernel.

Since ISRs execute in the context of an Ada task, an ISR can perform any semaphore service without disabling interrupts.

For DELETE_SEMAPHORE, since it is unable to detect if any task is waiting on the semaphore, it always assumes the semaphore is in use. Therefore, if DELETE_SEMAPHORE is called with DELETE_OPTION set to DELETE_OPTION_WARNING, it will never be deleted. On the other hand, if DELETE_SEMAPHORE is called with DELETE_OPTION set to DELETE_OPTION_FORCE, it does a dummy semaphore signal and waits 3.0 seconds before freeing the semaphore resources.

3.11.0.5 V_STACK

No differences.

3.11.0.6 V_XTASKING

The following tasking services are supported differently from the VADS MICRO version:

DISABLE_PREEMPTION
ENABLE_PREEMPTION

These services are layered on the following Ada Kernel services:

- TASK_DISABLE_PREEMPTION
- TASK_ENABLE_PREEMPTION

Refer to the previous section on the Solaris MT implementation of preemption in the Ada Kernel.

INSTALL_CALLOUT

Supports only the program events of EXIT_EVENT and UNEXPECTED_EXIT_EVENT.

OS_ID

Returns the Solaris thread ID.

RESUME_TASK
SUSPEND_TASK

These services are layered on the Ada Kernel services, TASK_RESUME, and TASK_SUSPEND. Refer to the previous section on the Solaris MT implementation of task suspend/resume in the Ada Kernel.

The following tasking services are not supported:

ALLOCATE_TASK_STORAGE	- returns NO_TASK_STORAGE_ID
CURRENT_TIME_SLICE	- returns 0.0
CURRENT_TIME_SLICING_ENABLED	- returns FALSE
GET_PROGRAM_KEY	- returns NO_ADDR
GET_TASK_STORAGE	- returns NO_ADDR
GET_TASK_STORAGE2	- returns NO_ADDR
SET_TIME_SLICE	- silently ignored
SET_TIME_SLICING_ENABLED	- silently ignored
START_PROGRAM	- returns NO_PROGRAM_ID

"How these curiosities would be quite forgott, did not such idle fellowes as i am putt them downe."

John Aubrey

Debugging in the Multithreaded Environment 4

4.1 Introduction

All of the normal debugger commands are available in the Multithreaded Ada environment along with a few additional ones.

4.2 Synchronous Operation

The Solaris MT debugger operates synchronously with the program. Either the program is stopped and the debugger can accept and execute commands, or the program is running and the debugger is waiting for an event to happen in the program. When the program is running, all of the lightweight processes (LWPs) are executing. This includes executing single step commands. Every time you single step your program (with `s`, `a`, `si` or `ai`), the debugger restarts each of the LWPs in the program. This can lead to surprising behavior, since the operating system may reschedule the set of LWPs onto the physical processors between the times the debugger stops and restarts the program.

When one of the LWPs hits a breakpoint, the operating system stops it and notifies all the other LWPs in the program. They also stop.

4.3 Debugger Signal Handling

Normally, the debugger intercepts all signals that are being delivered to the target program. However, since they are used by the Solaris threads layer, `SIGALRM`, `SIGLWP`, and `SIGWAITIN` are ignored by the debugger and not announced to the user.

4.4 Listing Ada Tasks

When the program is stopped, the `lt` command can be used to display the status of all of the Ada tasks. The task which hit the latest breakpoint or announced the latest signal is marked with an asterisk and its status is listed as `executing`. Note that in an application running under Solaris MT, several tasks can be executing concurrently. Only the task that announced the latest event is listed with the `executing` status, however.

Since the Solaris threads layer has idle threads, it is possible that, if you stop a program with Control-c, the currently executing task is one of these idle threads. These do not correspond to any Ada task. In this case, the executing task is listed along with all of the Ada tasks in the `lt` command, but `lt` will only say that it is a non-Ada task. At this point you can get into the context of an Ada task by using the `task` command.

There are some unusual effects that are possible when you select a new task with the `task` command. When you type a `cs` command, it shows that you are at the bottom of the call stack, but the frame number that appears at the left is not 1. This is because there are call stack frames that belong to the threads layer beneath the bottom Ada frame. If you want to descend to the bottom of the thread layer's call stack, you can type `cu 1` at this point. Typing a `cs` command now will provide you some additional information about what your program is doing.

Another unusual effect of selecting a new Ada task is that the debugger may not be able to determine the values of all of the registers. This is because not all register values are saved and restored by context switches either at the Ada or the threads layer.

The task status display of the `lt` command has been augmented to include the `thread_id` of the thread corresponding to the Ada task.

References

`lt` command, *SPARCompiler Ada Reference Guide*

`task` command, *SPARCompiler Ada Reference Guide*

Index