

POSIX Threads on HP-UX 11i

HP-UX 11i v2 update 2, release date September 2004



Intended Audience	2
Goals	2
Introduction	2
Thread Types	3
Operation Overview	3
Application behavior with MxN thread model	4
Controlling Kernel Concurrency	4
API Differences	4
Applications with mixed objects and libraries	5
Additional Mutex Enhancements	5
Multi-threaded Performance	6
Threads Tuning on HP-UX	7
Properties File	7
Environment Variables	8
Scope Options provided by the thread library	9
Properties File Scope Options	9
Environment variable scope options	9
Compile time scope options	10
Precedence	10
Tool Support for MxN	11
For more information	12

Intended Audience

This document is intended for the developers and performance experts who develop and tune multi-threaded applications using the POSIX threads implementation on HP-UX. A technical audience with exposure to multithreading environments is assumed.

Goals

The goal of this paper is to provide an overview of the thread model implementation on HP-UX 11i and describe the impact that each thread model has on application behavior and performance.

For additional information on multi-threaded programming in general [Thread Time](#) (ISBN 0-13-190067-6) by Scott Norton and Mark Dipasquale, and [Programming with POSIX Threads](#) (ISBN 0-20-163392-2) by David Butenhof are highly recommended.

Introduction

The multi-threaded programming model has been prevalent in the software development process for over a decade now. Threads provide a convenient programming model for handling concurrent activities. Multi-threading offers increased throughput, better response time, conservation of system resources and a natural programming structure. The performance benefits are realized through concurrent thread execution. Multi-threaded programs can exploit the power of multiprocessor systems and can often make the most efficient use of uniprocessor systems.

There are mainly three types of thread models used for multi-threaded programming: Mx1, 1x1, and MxN. The Mx1 (Many-to-One) model is a user space implementation of threads, with no involvement of the kernel. The main drawback of this model is its inability to exploit the multiprocessor capabilities of a system. The 1x1 (One-to-One) model is a kernel implementation of threads which has more thread management overhead and kernel resource utilization issues, especially with applications having a large number of threads. The MxN (Many-to-Many) model is a combination of 1x1 and Mx1 models and has the advantages of both models, while having increased complexity in the implementation.

HP-UX has supported both the Mx1 model (through the library `libcma`) and the 1x1 model (through the library `libpthread`) over the last several years. Although the requirement for MxN threading model was evident for a certain subclass of applications, HP did not initially introduce such a model. HP engineering staff analyzed the potential problems an inferior MxN implementation could cause to multi-threaded applications. Resources and time were spent on developing a thread implementation that can effectively address the needs of large multi-threaded applications, while providing the flexibility for users to choose the thread model that is best suited to their application with the same library implementation.

HP-UX 11i v1.6 introduced support for the MxN thread model allowing finer control over applications. The MxN model is a hybrid model that allows applications to multiplex user threads on top of kernel threads or to bind user threads to kernel threads (1x1 model) as necessary to suit program needs. This new implementation was designed to provide an effective means to address the needs of large multi-threaded applications, while providing the flexibility for users to choose the thread model best suited to their application with the same library implementation.

Thread Types

The POSIX thread standard defines two types of threads: `PTHREAD_SCOPE_SYSTEM` and `PTHREAD_SCOPE_PROCESS`. While the standard defines both types of threads, a compliant implementation is not required to provide both.

Threads created with `PTHREAD_SCOPE_SYSTEM` contend for resources with all other threads of the same scope in the system. This thread attribute is used to specify that the user thread is to be bound directly to a kernel-scheduled entity for its lifetime. These threads are also referred to as bound threads.

Threads created with `PTHREAD_SCOPE_PROCESS` contend for the resources only with other threads within the same process that were created with the same scope. This thread attribute is used to specify that the user thread is not to be bound to a specific kernel-scheduled entity. These threads are also referred to as unbound threads.

The scheduling policy and priority of `PTHREAD_SCOPE_PROCESS` threads is relative to other `PTHREAD_SCOPE_PROCESS` threads in the same process and is independent of `PTHREAD_SCOPE_SYSTEM` threads in the same process and `PTHREAD_SCOPE_PROCESS` threads in other processes. The scheduling policy and priority of `PTHREAD_SCOPE_SYSTEM` threads has system-wide scope and affects scheduling of `PTHREAD_SCOPE_SYSTEM` threads in other processes as well.

Each of the three threading models supports one or both of these thread types. The 1x1 model supports only `PTHREAD_SCOPE_SYSTEM` threads and the Mx1 model supports only `PTHREAD_SCOPE_PROCESS` threads. The MxN model supports both the types and allows the application to select one of them.

Operation Overview

The underlying kernel entity for a bound thread is a kernel thread (KT), also referred to as a Light Weight Process (LWP). A bound thread is always tied to a particular kernel thread and the scheduling of a bound thread is controlled by the kernel scheduler. Unbound threads are multiplexed over one or more kernel-scheduled entities by the user space scheduler. An unbound thread may move from one kernel-scheduled entity to another in its lifetime. The kernel-scheduled entities on which an unbound thread is scheduled can be considered as virtual processors. A virtual processor (VP) abstraction is used to schedule unbound threads. The virtual processor abstraction uses the underlying kernel entity called Scheduler Activation (SA). An SA is very similar to a kernel thread and is scheduled by the kernel scheduler just as any other kernel thread. The user space scheduler supports multiple virtual processor abstractions, with its associated run queues containing user threads, for a single process. It also supports a user space sleep/wakeup mechanism for unbound threads, real-time unbound thread scheduling, and suspension/resumption of unbound threads.

The MxN user space scheduler uses a parameter called concurrency level that controls the number of schedulable entities in the kernel. By default, the concurrency level is set to the number of processors in the machine. An application can provide a hint to the user space scheduler by using the Application Programming Interface (API), `pthread_setconcurrency(3T)`. This feature may be useful to applications that require better scheduling control for their threads.

SAs differ from kernel threads in their blocking characteristics. SA has the ability to provide replacement SA to the user space scheduler when the unbound thread running on a particular virtual processor performs a blocking operation in the kernel. The user space scheduler can replace the underlying SA of the virtual processor which is blocked, with this new SA and continue operations of the virtual processor. The kernel uses a lightweight upcall mechanism to communicate the blocking and unblocking events of scheduler

activations to the user space scheduler. A scheduler activation pool is maintained for quick replacement of SAs.

Application Behavior with MxN thread model

The POSIX standard provides the `pthread_attr_setscope(3T)` interface to control which type of thread is created. The standard does not specify a default contention scope, for maximum portability applications should always explicitly define thread attributes. When no contention scope is specified, the value used is implementation dependant.

On HP-UX 11.0 and 11i v1, the thread model used is the 1x1 model; all threads are created with the contention scope of `PTHREAD_SCOPE_SYSTEM`. With the introduction of the MxN model on HP-UX 11i v1.6, the default contention scope was defined as `PTHREAD_SCOPE_PROCESS`. This resulted in differences for applications migrating from older releases to the newer release. HP-UX 11i v2 update 2 changes the default contention scope to `PTHREAD_SCOPE_SYSTEM` improving compatibility for applications moving between HP-UX 11i v1 and 11i v2. Applications should explicitly define contention scope for maximum compatibility and portability.

Multi-threaded applications developed on HP-UX 11i v1, when run on HP-UX 11i v2 without recompilation, will run using the 1x1 threading model as on HP-UX 11i v1. Applications recompiled on HP-UX 11i v2 update 2, will use the MxN model with a default contention scope of `PTHREAD_SCOPE_SYSTEM`. When recompiled, if the application uses `pthread_attr_setscope(3T)` to set the scope `PTHREAD_SCOPE_PROCESS`, process scope thread will be created.

Multi-threaded applications built on HP-UX 11i v2, when run on HP-UX 11i v2 update 2 with and without recompilation, will use a default contention scope of `PTHREAD_SCOPE_SYSTEM`. If the application uses `pthread_attr_setscope(3T)` to set the scope `PTHREAD_SCOPE_PROCESS`, process scope threads will be created.

This behavior can be changed by using the properties file options, environment variables or compile time options which are discussed in the subsequent sections of this document.

Controlling Kernel Concurrency

An application can notify the library of its desired number of kernel entities (also called concurrency level) used for scheduling unbound threads, by calling the `pthread_setconcurrency(3T)` API. However, this is taken only as a hint and the actual level of concurrency provided with this function call is unspecified. The default concurrency level can be obtained calling the `pthread_getconcurrency(3T)` API provided that no call to `pthread_setconcurrency(3T)` API was made earlier.

API Differences

The behavior of some non-POSIX APIs, `sigstack(2)`, `sigaltstack(2)` and `sigspace(2)`, is undefined when called from unbound threads. These system calls should be called only from bound threads for them to work correctly.

The functions `pthread_pset_bind_np(3T)` and `pthread_launch_policy_np(3T)` are not supported for unbound threads. This behavior is documented in the HP-UX 11i man pages for these APIs. Gang scheduling is not supported for unbound threads. It is recommended for performance reasons that `PTHREAD_SCOPE_SYSTEM` threads be used for processor binding and not `PTHREAD_SCOPE_PROCESS` threads.

When an application calls the `sched_setparam()` or `sched_setscheduler()` system calls, all the kernel entities used for scheduling unbound threads in the process will be affected. It will not have any effect on kernel entities running bound threads.

The signal `SIGLWPTIMER` is used internally by the implementation to timeshare-unbound threads with the `SCHED_TIMESHARE` scheduling policy. This signal should not be used by the application.

The `STOP` and `CONTINUE` signals posted using `pthread_kill(3T)` to threads created with `PTHREAD_SCOPE_PROCESS` contention scope in the same process may not adhere to the job control semantics followed for `PTHREAD_SCOPE_SYSTEM` threads. Refer to the `signal(5)` man page for more information.

Users of the `pstat` interfaces and other system tools showing kernel scheduling entities should be aware of internal helper threads which may be created by the thread library. In addition, kernel entities used for scheduling unbound threads may be cached by `libpthread` and may also show up as kernel entities belonging to a process. An application should not make any assumptions about the number of kernel entities used by a multi-threaded process, even if the application is creating only bound threads.

Applications with mixed objects and libraries

Applications and libraries may be constructed using multiple contention scopes. It is also possible that an application may use libraries compiled on an earlier release of HP-UX, this is most common with third party libraries. Applications which create threads from multiple sources, both library and executable for example may encounter this situation.

An application compiled on HP-UX 11i v2 on HP 9000 (PA-RISC) systems which uses libraries compiled on HP-UX 11i v1 will use either the 1x1 or MxN model based on which object initializes or creates the first thread. If the executable calls `pthread_create(3T)` or `pthread_attr_init(3T)` first then the runtime will behave as if the entire application, including the library, was compiled on HP-UX 11i v2 update 2. The application will be able to create both system scope (bound) and process scope (unbound) threads with a default of system scope, `PTHREAD_SCOPE_SYSTEM`.

If the library, compiled on HP-UX 11i v1, calls `pthread_create(3T)` or `pthread_attr_init(3T)` first then the runtime will behave as if compiled on HP-UX 11i v1. The entire application will utilize the 1x1 model, and process scope (unbound) threads will not be available. Calls to `pthread_attr_setscope(3T)` requesting `PTHREAD_SCOPE_PROCESS` threads will not have any effect.

Similar situations can occur when only a portion of the application (or library) is compiled using any of the compile time defines (i.e. `-DPTHREAD_COMPAT_MODE`). If consistent behavior is needed, all modules must be compiled with the same setting. In situations where it is not possible to recompile all modules the user can set the corresponding properties file option or environmental variable. For example, if the application needs the default scope as `PTHREAD_SCOPE_SYSTEM` and process scope threads to be created when specified through the attribute, add `"default_scope_system 1"` to the properties file or set the environment variable `PTHREAD_DEFAULT_SCOPE_SYSTEM` to 1. These options are described in more detail in the subsequent sections.

Additional Mutex Enhancements

HP-UX 11i v1.6 introduced additional mutex performance enhancements. The thread library supports a mechanism to enable "Release" mode for mutexes. "Release" mode is a feature that can be used to enhance the performance of the mutex when it is highly contended.

Without this mode, all process shared mutexes use a "Handoff" mode mechanism to operate on mutexes whereby a mutex lock is never really released, but rather handed over to the highest priority waiter, if any. This is done to adhere to standards which mandate that a low priority thread should not acquire the mutex when there is a higher priority real time thread waiting for the same mutex. This protocol does not allow another running thread to

acquire and release the mutex (during the time required for waking up the thread after changing the owner of mutex to that thread), and let it make faster forward progress.

Process private mutexes will continue to use the "Release" mode by default, but they will switch to "Handoff" mode when real time threads attempt to acquire the mutex lock.

The "Handoff" mode can be disabled entirely for both private and shared mutexes by using one of the following options:

1. Using the API `pthread_mutex_disable_handoff_np(3T)`, which disables "Handoff" mode for all mutexes in the application.
2. Using the API `pthread_mutexattr_disable_handoff_np(3T)`, which disables "Handoff" mode for the specified mutex.
3. Setting the environment variable, `PTHREAD_DISABLE_HANDOFF` to either 1 or ON which disables "Handoff" mode for all mutexes in the application.

NOTE: Applications will not adhere to real-time semantics if this behavior is enabled, however it can avail the performance benefits provided by this functionality.

Multi-threaded Performance

The performance of a multi-threaded program largely depends on the nature and design of the application. The thread model and type chosen should be based on the nature of the application.

The main advantage of the MxN model is its ability to support both bound and unbound threads and thus provide flexibility for optimizing a program to meet its specific needs. This gives the MxN model all the advantages of both the 1x1 and Mx1 thread models and eliminates some of the disadvantages with these models. In addition to providing concurrency and parallelism, some of the other advantages of the MxN model are faster thread management operations and conservation of system resources.

The design and implementation of the HP-UX MxN model considered the complexities involved in the interaction between the user space and the kernel scheduler. The scheduler activation model effectively addresses these complexities to gain better performance for several types of multi-threaded applications. The scheduler activation model uses a lightweight upcall mechanism to facilitate effective communication between the user space and the kernel scheduler. The HP-UX implementation also uses a scheduler activation cache to speed up the controlled replacement of blocked kernel entities. The self-scheduling virtual processor design eliminates potential lock contention issues in the implementation. The MxN implementation does not affect the behavior of single-threaded applications linked with the threads library.

Several parallel programming models like the "master-slave model" or the "work-crew model" are available for program design. These programming models generally use thread pools with fewer kernel threads to achieve better performance, with additional application complexity. The MxN model, however, can achieve a higher throughput than the 1x1 model for today's application design and workloads without additional complexity.

The `PTHREAD_SCOPE_PROCESS` feature of the MxN threads implementation can improve performance for a subclass of multi-threaded applications. Applications which use a large number of threads benefit most. Applications which have very few threads or are computation-bound benefit the least. The design minimizes the user space scheduling overhead for applications having very few threads; the majority of such applications do not notice any difference in their performance. However, some applications having very few threads can be hurt by the user space scheduling overhead. Applications which are extremely compute bound are one class of application where this has been observed. In this case, it is recommended that the application use system scope (bound) threads created using `PTHREAD_SCOPE_SYSTEM`.

Several benchmarks assess the performance benefits of the MxN model. Some of the results are given below. The measurements are based on a 4-way machine.

- SPECjbb: This is a Java-based compute-intensive benchmark with a single thread per processor model. The MxN model, as expected, does not give any performance benefits to such applications. The throughput achieved using the MxN model is comparable with that of the 1x1 model. However, the 1x1 model is best suited to such applications, especially on larger boxes, to avoid any user space scheduling overhead.
- VolanoMark: This is a Java server benchmark characterized by long-lasting network connections and high thread counts. The MxN model shows a 15% improvement over the 1x1 model.
- Ping-Pong: Sun published the ping-pong benchmark, which is used to compare their implementations of the 1x1 and MxN models. The benchmark consists of pairs of threads (simulating a game) in lock-step synchronization. With an increasing number of threads, the HP-UX MxN model with a higher concurrency level shows a performance improvement of about 30% over the 1x1 model.
- Dining Philosophers: This is a performance benchmark of academic interest and involves intense synchronization between several threads trying to acquire a global resource. The benchmark implements a solution using global condition variables and mutex locks. The MxN model shows a clear advantage over the 1x1 model for such applications, with a performance improvement of nearly 3 times.

Threads Tuning on HP-UX

Fine tuning the application threading profile can provide further performance improvements beyond normal compiler optimizations. HP-UX provides various methods for tuning and controlling the application threading environment.

Properties File

Starting with HP-UX 11i v1.6, users have the flexibility to use a libpthread properties file to specify various tunables for the library, to help in optimizing their applications without making any source code changes. The default path name of the properties file is `/usr/lib/libpthread.properties`. Please note that this file will have to be explicitly created by the user. If the application requires the pthread library to read the tunables from the properties file, the environment variable `PTHREAD_TUNE` must be set to 1, or ON. The user, through the environment variable `PTHREAD_PROPERTY_FILE` can specify a different location for the properties file. This provides additional flexibility to support application specific needs on the same system.

In the properties file, lines beginning with # are comments. For non-comment lines, the first and second words are extracted. The first word is expected to be the name of one of the tunables, and the second word is expected to be the value of that variable. The syntax of properties file tunable is as follows:

```
<tunable name> <value>
```

If the `<tunable name>` doesn't match any of the known tunables, or `<value>` is not a legitimate number or the number is not in the specified range, it will not have any impact.

The following are the tunable parameters supported by the libpthread properties file.

<code>all1x1</code>	This tunable when set to 1, indicates to the library that all threads created by the application have to be bound irrespective of the contention scope specified in the application. This is equivalent to compiling the application with <code>-DPTHREAD_COMPAT_MODE</code> .
<code>concurrency</code>	This tunable when set to <code><value></code> , indicates to the library that the number specified by <code><value></code> should be used as the concurrency level of the application. This is equivalent to the application calling the <code>pthread_setconcurrency(3T)</code> API. The

default value is same as the number of processors on the system.

It can be set to a positive value, not exceeding `_SC_THREAD_THREADS_MAX`

<code>timeslice</code>	<p>This tunable when set to <code><value></code>, indicates to the library that the number in milliseconds specified by <code><value></code> be used as the duration for which a <code>SCHED_TIMESHARE</code> unbound thread runs before it does an involuntary context switch. This is equivalent to the application calling <code>pthread_settimeslice_np(3T)</code> API.</p> <p>It can be set to a positive value (in milliseconds), not exceeding 999. The default value is 200 milliseconds.</p>
<code>mxnfromcompatmode</code>	<p>This tunable when set to 1, indicates to the library that the application can create unbound threads in the 1x1 compatibility mode by setting the <code>PTHREAD_SCOPE_PROCESS</code> attribute. Note that if both <code>"mxnfromcompatmode"</code> and <code>"all1x1"</code> are set, then <code>mxnfromcompatmode</code> overrides <code>all1x1</code>. The default value is -1.</p>
<code>maxblocklimit</code>	<p>This tunable controls the number of outstanding blocking upcalls that can be pending on a particular VP at any point in time. The default value is 5.</p>

For example, to run an MxN application in 1x1 mode, the following needs to be done:

Create a properties file. (The default pathname is `/usr/lib/libpthread.properties`. If you want an application specific file, set `PTHREAD_PROPERTY_FILE` with `<your path name>` e.g., `PTHREAD_PROPERTY_FILE=/tmp/my.property`)

Add `"all1x1 1"` to the properties file

```
$(PTHREAD_TUNE=1; <Invoke application>)
```

The following is an example of the `libpthread` properties file:

```
concurrency 8
timeslice 200
all1x1 1
maxblocklimit 10
```

Environment Variables

The following environment variables are supported to provide flexibility in tuning multi-threaded applications

<code>PTHREAD_DISABLE_HANDOFF</code>	<p>This variable is used to disable the handoff mode mutex behavior. This may improve application performance.</p> <p>Valid values: ON, 1</p>
<code>PTHREAD_COMPAT_MODE</code>	<p>This variable is used to enable the 1x1 compatibility mode.</p> <p>Valid values: ON, on, 1</p>
<code>PTH_CONC_LEVEL</code>	<p>This variable is used to set the concurrency level of the process</p> <p>Valid values: A nonzero value.</p>
<code>PTHREAD_TUNE</code>	<p>This variable when set causes the library to read the tunable values from the property file. The default file name is <code>/usr/lib/libpthread.properties</code></p> <p>Valid values: ON, on, 1</p>
<code>PTHREAD_PROPERTY_FILE</code>	<p>This variable is used to specify the name of the <code>libpthread</code> tunable property file. This will be read only when <code>PTHREAD_TUNE</code> is also set.</p> <p>Valid values: A readable filename that has the <code>libpthread</code> tunable parameters.</p>

Scope Options provided by the thread library

Controlling thread scope can also be used to improve performance; users are given the flexibility to control the scope of threads created on HP-UX 11i v2 update 2. New options can be set in the properties file, the environment variables and at compile time. This helps multi-threaded application developers to test the applications with various scope settings for the threads without actually modifying the source code. It can be used to determine the most efficient thread model for any particular application.

Properties file scope options

The following scope options are available in the properties file for HP-UX 11i v2 update 2 and later. The options are listed in order of decreasing precedence, those listed first override those listed later.

<code>force_scope_system</code>	If this option is set to 1, the application will get only system scope threads even when <code>PTHREAD_SCOPE_PROCESS</code> is specified by <code>pthread_attr_setscope(3T)</code> . If this tunable is set, the application gets the 1x1 model behavior. Default value is -1
<code>force_scope_process</code>	If this option is set to 1, the application will get only process scope threads even when <code>PTHREAD_SCOPE_SYSTEM</code> is specified by <code>pthread_attr_setscope(3T)</code> . Default value is -1
<code>default_scope_system</code>	If this option is set to 1, the application will get system scope threads by default and if <code>PTHREAD_SCOPE_PROCESS</code> is specified by <code>pthread_attr_setscope(3T)</code> , process scope thread will be created. Default value is -1
<code>default_scope_process</code>	If this option is set to 1, the application will get process scope threads by default. However, if <code>PTHREAD_SCOPE_SYSTEM</code> is specified by <code>pthread_attr_setscope(3T)</code> , system scope thread will be created. Default value is -1

Note: If any of the above four new options are set, `mxnfromcompatmode` and `all1x1` will not have any effect.

Example: If an application requires process scope threads to be created by default, then the following entry should be included in the properties file:

```
default_scope_process 1
```

Environment variable scope options

The following environment variables are available to control the thread contention scope at run time on HP-UX 11i v2 update 2 and later. The options are listed in order of decreasing precedence, those listed first override those listed later.

<code>PTHREAD_FORCE_SCOPE_SYSTEM</code>	This variable is used to specify that the application needs only system scope threads even when <code>PTHREAD_SCOPE_PROCESS</code> is specified by <code>pthread_attr_setscope(3T)</code> . When this variable is set, the application gets the 1x1 behavior. Valid values: ON, on, 1
<code>PTHREAD_FORCE_SCOPE_PROCESS</code>	This variable is used to specify that the application needs only process scope threads even when <code>PTHREAD_SCOPE_SYSTEM</code> is specified by <code>pthread_attr_setscope(3T)</code> . Valid values: ON, on, 1

<code>PTHREAD_DEFAULT_SCOPE_SYSTEM</code>	This variable is used to specify that the application needs system scope threads by default and if <code>PTHREAD_SCOPE_PROCESS</code> is specified by <code>pthread_attr_setscope(3T)</code> , process scope threads will be created. Valid values: ON, on, 1
<code>PTHREAD_DEFAULT_SCOPE_PROCESS</code>	This variable is used to specify that application needs process scope threads by default and system scope threads if <code>PTHREAD_SCOPE_SYSTEM</code> is specified by <code>pthread_attr_setscope(3T)</code> . Valid values: ON, on, 1

If any of the above mentioned new environment variables are set along with `PTHREAD_COMPAT_MODE`, `PTHREAD_COMPAT_MODE` takes precedence.

Example: If an application needs to have the default scope as "process scope":

```
PTHREAD_DEFAULT_SCOPE_PROCESS=1; ./my_app
```

This will ensure that the setting is in effect only for a particular application.

Compile time scope options

If the application is compiled with `PTHREAD_COMPAT_MODE` by specifying with `-D` option or defining it in one of the application header files before including `pthread.h`, `PTHREAD_SCOPE_SYSTEM` will be forced for the creation of all the threads in the application and the application gets the 1x1 behavior.

The following additional compile time options control the thread contention scope on HP-UX 11i v2 update 2 and later. The options are listed in order of decreasing precedence, those listed first override those listed later.

<code>PTHREAD_FORCE_SCOPE_SYSTEM</code>	This option is used to specify that the application needs only system scope threads even when <code>PTHREAD_SCOPE_PROCESS</code> is specified by <code>pthread_attr_setscope(3T)</code> . This allows the application to get the HP-UX 11i v1 (1x1) model behavior.
<code>PTHREAD_FORCE_SCOPE_PROCESS</code>	This option is used to specify that the application needs only process scope threads even when <code>PTHREAD_SCOPE_SYSTEM</code> is specified by <code>pthread_attr_setscope(3T)</code> .
<code>PTHREAD_DEFAULT_SCOPE_PROCESS</code>	This option is used to specify that the application needs process scope threads by default and if <code>PTHREAD_SCOPE_SYSTEM</code> is specified by <code>pthread_attr_setscope(3T)</code> , system scope threads will be created.

Important Note: The applications compiled on HP-UX 11i v2 update 2 with any of the above three compile options, will not run on HP-UX 11i v2.

The `PTHREAD_COMPAT_MODE` compile option that is supported on HP-UX 11i v2 is supported on HP-UX 11i v2 update 2. An application compiled with this option on HP-UX 11i v2 update 2 will run on HP-UX 11i v2. If any of the above mentioned new compile options are defined along with `PTHREAD_COMPAT_MODE`, then the new compile options will not have any effect.

Precedence

As specified in the earlier sections, multiple forms of altering the contention scope are available: compile time defines, properties files and runtime environment. If more than one is enabled, only one will be honored.

The scope options provided in the runtime environmental variables override the scope options set in the properties file. The properties file will override scope options defined at the

compile time. Within each method of defining the scope there is a further precedence. "Force" options will have precedence over "default" options. Options setting system scope have precedence over process scope options.

Tool Support for MxN

The HP WDB debugger is an HP-supported implementation of the GDB debugger. It supports source-level debugging of object files written in HP C, HP aC++, Fortran 90, and FORTRAN77 on HP-UX Release 11.0 and later. HP WDB also includes full support for debugging multi-threaded applications under both the 1x1 and MxN thread models. Information on the latest release of HP WDB can be found at www.hp.com/go/wdb.

HP Caliper is a general-purpose performance analysis tool for applications on HP-UX systems. HP Caliper allows you to understand the performance of your program and to identify ways to improve its run-time performance. HP Caliper works with any Itanium®-based binary and does not require your applications to have any special preparation to enable performance measurement. HP Caliper can be used for both standard application profiling and PBO-based compile-time optimization on both threaded and non-threaded applications. More information on HP Caliper, including download information, is available at www.hp.com/go/caliper.

Another useful tool when debugging and profiling threaded applications is Visual Threads. It can be used to automatically diagnose common problems associated with multithreading, including deadlock and thread usage errors. It can also be used to monitor the thread-related performance of the application, helping you to identify bottlenecks or locking granularity problems. It is a unique debugging tool because it can be used to identify problem areas even if an application does not show any specific problem symptoms. Details on installing and running Visual Threads is available at www.hp.com/go/visualthreads

For more information

www.hp.com/dspp

docs.hp.com

[Thread Time](#) (ISBN 0-13-190067-6) by Scott Norton and Mark Dipasquale

[Programming with POSIX Threads](#) (ISBN 0-20-163392-2) by David Butenhof

© 2003 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Itanium is a trademark or registered trademark of Intel Corporation in the U.S. and other countries and is used under license.

XXXX-XXXXEN, 07/2003

