Debugging threads with HP Wilde Beest Debugger

HP Part Number: 5992-4663
Published: September 2008
Edition: 1.0

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

U.S. Government License Proprietary computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice Copyright © 2008 Hewlett-Packard Development Company, L.P.Reproduction, adaptation, or translation of this document without prior written permission is prohibited, except as allowed under the copyright laws.

Trademark Notices

UNIX is a registered trademark of The Open Group.

Intel and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Table of Contents

List of Tables

About This Document

This whitepaper describes the various commands and options available in HP Wilde-Beest Debugger (WDB) to debug threads in programs. In addition, this paper addresses the most common thread programming issues along with the various thread-debugging commands in HP WDB.

Intended Audience

This document targets the developers who want to use HP WDB to debug their threaded applications developed in C and C++. The document also intends to be a useful reference for readers who want to know about the various thread-debugging features available in HP WDB.

Typographic Conventions

This document uses the following typographical conventions:

`%`, `$`, or `#`	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells. A number sign represents the superuser prompt.
audit(5)	A manpage. The manpage name is *audit*, and it is located in Section 5.
`Command`	A command name or qualified command phrase.
`Computer output`	Text displayed by the computer.
Ctrl+x	A key sequence. A sequence such as **Ctrl+x** indicates that you must hold down the key labeled **Ctrl** while you press another key or mouse button.
`ENVIRONMENT VARIABLE`	The name of an environment variable, for example, `PATH`.
[ERROR NAME]	The name of an error, usually returned in the `errno` variable.
Key	The name of a keyboard key. **Return** and **Enter** both refer to the same key.
Term	The defined use of an important word or phrase.
`User input`	Commands and other text that you type.
`Variable`	The name of a placeholder in a command, function, or other syntax display that you replace with an actual value.
[]	The contents are optional in syntax. If the contents are a list separated by `|`, you must choose one of the items.

{}	The contents are required in syntax. If the contents are a list separated by |, you must choose one of the items.
...	The preceding element can be repeated an arbitrary number of times.
◎	Indicates the continuation of a code example.
|	Separates items in a list of choices.
WARNING	A warning calls attention to important information that if not understood or followed will result in personal injury or nonrecoverable system problems.
CAUTION	A caution calls attention to important information that if not understood or followed will result in data loss, data corruption, or damage to hardware or software.
IMPORTANT	This alert provides essential information to explain a concept or to complete a task
NOTE	A note contains additional information to emphasize or supplement important points of the main text.
TIP	A tip contains information that is a helpful hint to solving an issue or a problem.

Related Information

The HP WDB documentation is available at the following location:

`/opt/langtools/wdb/doc/`

Table 1 lists the documentation available for WDB.

Table 1 Documentation for HP WDB

Document	Location
Debugging with GDB	`/opt/langtools/wdb/doc/gdb.pdf`

GDB Quick Reference Card	`/opt/langtools/wdb/doc/refcard_a4.pdf`
	`/opt/langtools/wdb/doc/refcard_a3.df`
	`/opt/langtools/wdb/doc/refcard.pdf` (Letter Format)

Getting Started with WDB	`/opt/langtools/wdb/doc/html/wdb/C/GDBtutorial.html`
WDB Online Help	`/opt/langtools/wdb/doc/index.html`
HP WDB GUI Documentation	`/opt/langtools/wdb/doc/html/wdbgui/C/`
GDB manpage	`gdb(1)`

For the most current WDB documentation, see the *HP WDB technical resources* website at:

http://www.hp.com/go/wdb

End of Reading

At the end of reading, the reader gains a fair understanding of the following:

- Quick overview on thread programming
- Basics of advanced thread-debugging using HP WDB
- Various modes of advanced thread debugging
- Brief description of thread-related conditions
- Commands available to debug common thread-related conditions
- Best practices to avoid thread-related issues

What are threads?

Threads are subsets of 'Process', which aid in accelerating the execution of any task. The usage of threads increases efficiency that results from the intended concurrency in the threaded-programming practice. Threads share the same resources as Process and hence do not have resource overheads. This property of threads is important when processing speed becomes a criterion to measure efficiency and if programmers use more than one thread to complete a single process.

Today's hardware comes with multiple processors to support enhanced speed in processing. Multi-threaded programs which execute on multi-processor systems and multi-core systems make the best use of Parallelism that the hardware offers.

Understanding complexity in thread programming

The concurrency of threads and the parallelism of the hardware jointly contribute to the processing speed of programming applications. With threaded programs, ever on an increase in the industry, it becomes essential to understand the underlying complexity in implementing the concept. In addition, this understanding helps you interpret the results of debugging threaded programs.

The following are the most important concepts that attribute to the complexity in threaded programming:

- Deadlock
- Race condition
- Priority inversion

Mutual exclusion (mutex) is a method which ensures that the threads share program resources systematically, thereby avoid unintended modification of the data in shared variables. Program segments attach locks to shared resources. This ensures mutual exclusion. Improper mutex lock-unlock in threaded applications could result in a deadlock condition which stops the program execution completely.

Race condition arises when shared data or resources are not accessed in any particular order thus resulting in inconsistent data in some instances. This possibility occurs when you write code segments without ensuring serial access to shared resources. Such threaded programs with inconsistent and random access to shared variables contribute to the complexity involved in debugging threads.

In addition, when a high-priority thread waits for a lock which a low-priority thread holds, the priority inversion here results in lesser execution efficiency.

Thread programs, prone to such complexities, are subject to conditions or events that reduce efficiency or increase the potential for errors.

Common conditions or events in thread programming

The following are the most common conditions or events in thread programming that could lead to errors:

1. The thread attempts to acquire a non-recursive mutex that it currently holds.
2. The thread attempts to unlock a mutex or a read-write lock that it has not acquired.
3. The thread waits on a mutex or a read-write lock that is held by a thread with a different scheduling policy.
4. Different threads non-concurrently wait on the same condition variable, but with different associated mutexes.
5. The thread terminates execution without unlocking the associated mutexes or read-write locks.
6. The thread waits on a condition variable for which the associated mutex is not locked.
7. The thread terminates execution, and the resources associated with the thread continue to exist in the application because the thread has not been joined or detached.
8. The thread uses more than the specified percentage of the stack allocated to the thread.
9. The number of threads waiting on any pthread object exceeds the specified threshold number.

HP Wildebeest Debugger (WDB) offers advanced thread debugging features to support debugging of threaded applications and detection of these conditions.

Introducing the HP WDB Debugger

HP Wildebeest Debugger (WDB) is an HP-supported implementation of the open source debugger GDB.

HP WDB enables you to debug threaded programs written in HP C and HP aC++ on Itanium®-based systems running HP-UX 11i v2 or HP-UX 11i v3, and HP 9000 systems running HP-UX 11i v2, or HP-UX 11i v3 operating systems.

The main features related to thread-debugging in HP WDB are as follows:

- Enable and disable a thread
- Enable and disable advanced thread-debugging feature
- Display the stack trace of the current thread
- View information on the state of pthread primitives such as mutexes, read-write locks, and conditional variables
- Detect most thread-related conditions that are potential causes for errors in thread programming, through advanced thread-debugging feature
- Debug threads interactively after you attach GDB to a process

Thread Debugging Support in HP WDB

HP WDB provides thread-debugging support for kernel, user, and MxN threads. You can exclusively disable or enable specific thread execution. Advanced thread debugging support in HP WDB enables you to view information on pthread primitives and detect certain thread-related conditions.

NOTE: WDB supports pthread Parallelism, but it does not support compiler-generated parallelism like parallelism with Directives.

Support for enabling and disabling specific threads

If you suspect that a specific thread causes problems when you debug a multi-threaded application, HP WDB allows you to suspend the execution of all other threads in the application and debug this thread exclusively.

`thread disable <thread-no>`

The `thread disable <thread-no>` command prevents the specified threads from running until they are enabled again using the `thread enable <thread-no>` command.

`thread enable <thread-no>`

The `thread enable <thread-no>` command enables the specified thread to run when you enter the `continue` or `step` command. All threads are in the enabled state by default.

To disable a thread, execute the following command:

```
(gdb) thread disable 1
warning: ATTENTION!! Disabling threads may result in
deadlocks in the program. Disabling thread 1
```

To enable a thread, execute the following command:

```
(gdb) thread enable 1
Enabling thread 1
```

Back trace support for thread debugging

The following commands are available as backtrace support for thread debugging:

`bt`

The `bt`command provides the stack trace of the current thread in execution or the thread that accepts the signal in core files.

`thread apply all bt`

The `thread apply all bt` command enables you to display the backtrace of all threads. The `bt` command provides the stack trace of only the current thread under execution.

`backtrace_other_thread`

The `backtrace_other_thread` command prints the backtrace of all stack frames for a thread with stack pointer `SP`, program counter `PC`and address of `gr32` in the backing store`BSP`. This command enables you to view the stack trace when the stack is corrupted. When using this command, you must ensure that the `SP`,`PC` , and `BSP` values are valid.

The syntax for the `backtrace_other_thread` command is as follows:

```
backtrace_other_thread SP PC BSP
```

For PA RISC systems, the command is as follows:

```
backtrace_other_thread SP PC
```

This command prints backtrace of all stack frames for a thread with stack pointer SP and program counter PC.

Debugging programs with multiple threads

In some operating systems, such as HP-UX, a single program may have more than one thread of execution. The precise semantics of threads differ from one operating system to another, but in general, the threads of a single program are akin to multiple processes - except that they share one address space (that is, they can all examine and modify the same variables). On the other hand, each thread has its own registers and execution stack, and private memory.

GDB provides these facilities for debugging multi-thread programs:

- Automatic notification of new threads
- Thread-specific breakpoints

WARNING! These facilities are not yet available on every GDB configuration where the operating system supports threads. If your GDB does not support threads, these commands have no effect. For example, a system without thread support shows no output from `info threads', and always rejects the thread command, like this:

```
((gdb)) info threads
((gdb)) thread 1

Thread ID 1 not known. Use the "info threads" command to
see the IDs of currently known threads.
```

The GDB thread debugging facility allows you to observe all threads while your program runs - but whenever GDB takes control, one thread in particular is always the focus of debugging. This thread is called the current thread. Debugging commands show program information from the perspective of the current thread.

Whenever GDB detects a new thread in your program, it displays the target system's identification for the thread with a message in the form [`New systag`]. `systag` is a thread identifier whose form varies depending on the particular system.

For debugging purposes, GDB associates its own thread number - always a single integer - with each thread in your program.

Following commands are used to debug multi-threaded programs:

- `thread threadno`, a command to switch among threads
- `info threads`, a command to inquire about existing threads
- `thread apply [threadno] [all] args`, a command to apply a command to a list of threads

`info threads`

Display a summary of all threads currently in your program. GDB displays for each thread (in this order) :

1. the thread number assigned by GDB
2. the target system's thread identifier (systag)
3. the current stack frame summary for that thread

An asterisk `*` to the left of the GDB thread number indicates the current thread.

For example,

```
((gdb)) info threads
3 process 35 thread 27 0x34e5 in sigpause ()
2 process 35 thread 23 0x34e5 in sigpause ()
* 1 process 35 thread 13 main (argc=1, argv=0x7ffffff8)
at threadtest.c:68
```

On HP-UX systems:

For debugging purposes, GDB associates its own thread number - a small integer as - signed in thread-creation order - with each thread in your program.

Whenever GDB detects a new thread in your program, it displays both GDB's thread number and the target system's identification for the thread with a message in the form [`New systag`]- `systag` is a thread identifier whose form varies depending on the particular system. For example, on HP-UX, you see

```
[New thread 2 (system thread 26594)]
```

when GDB notices a new thread.

`thread threadno`

Make thread number `threadno` the current thread. The command argument `threadno` is the internal GDB thread number, as shown in the first field of the `info threads` display. GDB responds by displaying the system identifier of the thread you selected, and its current stack frame summary:

```
((gdb)) thread 2
[Switching to thread 2 (system thread 26594)]
0x34e5 in sigpause ()
```

As with the [`New...`] message, the form of the text after `Switching to` depends on your system's conventions for identifying threads.

```
thread apply [threadno] [all] args
```

The thread apply command allows you to apply a command to one or more threads. Specify the numbers of the threads that you want affected with the command argument `threadno`. The `threadno` is the internal GDB thread number, as shown in the first field of the `info threads` display. To apply a command to all threads, use `thread apply all args`.

Whenever GDB stops your program, due to a breakpoint or a signal, it automatically selects the thread where that breakpoint or signal happened. GDB alerts you to the context switch with a message of the form [`Switching to systag`] to identify the thread.

NOTE: On HP-UX 11.x, debugging a multi-thread process can cause a deadlock if the process is waiting for an NFS-server response. A thread can be stopped while asleep in this state, and NFS holds a lock on the `rnode` while asleep.

To prevent the thread from being interrupted while holding the `rnode`lock, make the NFS mount non-interruptible with the `nointr` flag. See `mount(1)`.

On HP-UX systems, you can control the display of thread creation messages. Following commands are used to control the display of thread creation:

```
set threadverbose on
```

Enable the output of informational messages regarding thread creation. The default setting is on. You can set it to `off` to stop the display of messages.

```
set threadverbose off
```

Disable the output of informational messages regarding thread creation. The default setting is `on`. You can set it to `on` to display messages.

```
show threadverbose
```

Display whether `set threadverbose` is `on` or `off`.

When your program has multiple threads, you can choose whether to set breakpoints on all threads, or on a particular thread.

```
break linespec thread threadno
break linespec thread threadno if ...
```

`linespec` specifies source lines; there are several ways of writing them, but the effect is always to specify some source line.

Use the qualifier `thread threadno` with a breakpoint command to specify that you only want GDB to stop the program when a particular thread reaches this breakpoint. `threadno` is one of the numeric thread identifiers assigned by GDB, shown in the first column of the `info threads` display.

If you do not specify `thread threadno` when you set a breakpoint, the breakpoint applies to all threads of your program.

You can use the thread qualifier on conditional breakpoints as well; in this case, place `thread threadno` before the breakpoint condition, like this:

```
((gdb)) break frik.c:13 thread 28 if bartab > lim
```

Whenever your program stops under GDB for any reason, all threads of execution stop, not just the current thread. This allows you to examine the overall state of the program, including switching between threads.

Conversely, whenever you restart the program, all threads start executing. This is true even when single-stepping with commands like `step` or `next`.

Moreover, in general other threads stop in the middle of a statement, rather than at a clean statement boundary, when the program stops.

You might even find your program stopped in another thread after continuing or even single-stepping. This happens whenever some other thread runs into a breakpoint, a signal, or an exception before the first thread completes the action you requested.

Advanced Thread Debugging Support

Advanced thread debugging support is available for multi-threaded applications running on HP-UX 11iv2, or HP-UX 11iv3.

HP WDB 5.5 and later versions provide advanced thread debugging features to display extended information on the state of pthread primitives such as mutexes, read-write locks and conditional variables.

Enabling and Disabling Advanced Thread Debugging Features

The Advanced Thread Debugging features are available as options to the `set thread-check` command. The syntax for the set thread-check command is as follows:

```
set thread check
```

The `set thread-check` command enables or disables advanced thread de-bugging. This feature is `off` by default. The `set thread-check` command must be enabled prior to running the application under the debugger, to force the underlying runtime system to collect information on pthread primitives.

The advanced thread debugging features are available only if the `set thread-check` command is set to `on`.

Prerequisites for Advanced thread debugging feature in HP WDB

- HP-UX 11i v2 and later versions of OS on both PA-RISC and Integrity systems support the advanced thread debugging features.
- The thread debugging feature depends on the availability of the dynamic linker B.11.19 and later versions.
- The advanced thread debugging commands work only if `thread-check` is set to `on`.

- Advanced thread-debugging requires the `pthread` tracing library. The `pthread` tracer library is available by default in systems running on HP-UX 11i v2 or later. HP WDB 5.5 and later versions support enhanced thread debugging. The installation scripts for HP WDB 5.5 and later versions of the debugger automatically add links at `/opt/langtools/lib/` to replace the standard `libpthread` library with `libpthread` tracer library at run time.
- HP WDB uses `librtc.sl` to enable thread debugging support. If the debugger is available in a directory other than the default `/opt/langtools/bin` directory, use the environment variable, `LIBRTC_SERVER`, to export the path of the appropriate version of `librtc.sl`.
- For PA-RISC 32 bit applications, enable the dynamic path look-up for advanced thread debugging.
- The `chatr +rtc` feature requires linker version B.11.66 and later on HP 9000 systems, and linker version B.12.51 and later on Integrity systems.

NOTE: To enable dynamic library path look-up for advanced thread debugging, enter the following command at HP-UX prompt:

```
# chatr +s enable <PA32-bitApp>
```

This command automatically enables dynamic library path look-up.

Modes of Thread debugging in HP WDB

HP WDB offers three modes of Thread debugging:

- Interactive Mode
- Batch Mode
- Attach Mode
- The +check Mode

Thread-debugging in Interactive mode

Interactive mode of thread debugging is available for multi-threaded applications running on HP-UX 11iv2, or HP-UX 11iv3.

Using thread-debugging feature in HP WDB

Complete the following steps to use the thread-debugging feature in interactive mode:

1. Compile the program with –mt option to include threads in compilation:

   ```
   $ cc –mt –o a.out filename.c
   ```

2. Navigate to the path where `gdb` is available.
3. Enter the following command to invoke `gdb`:

   ```
   $ ./gdb
   ```

4. Invoke the executable that you want to debug:

   ```
   (gdb) file <Complete path of the executable or name of the executable>
   ```

5. Enable thread check along with the specific option as required.

   ```
   (gdb) set thread-check [option][on|off]
   ```

6. Execute the file with the following command:

   ```
   (gdb) run <Name of the executable>
   ```

The thread-check command

The advanced thread debugging features can be enabled only if the `set thread-check[on]` command is enabled. The following advanced thread debugging options are available for the `set thread-check` command:

- `recursive-relock [on|off]`
- `unlock-not-own [on|off]`
- `mixed-sched-policy [on|off]`
- `cv-multiple-mxs [on|off]`
- `cv-wait-no-mx [on|off]`
- `thread-exit-own-mutex [on|off]`
- `thread-exit-no-join-detach [on|off]`
- `stack-util [num]`
- `num-waiters [num]`

NOTE: By default all these options are turned on if you set the command `set thread-check on`.

Debugging common thread-programming problems

Problem: The thread attempts to acquire a non-recursive mutex that it currently has control.

Consider the following scenario:

Function 1 locks a non-recursive mutex and calls Function 2 without releasing the lock object. If Function 2 also attempts to acquire the same non-recursive mutex, the scenario results in a deadlock. In effect, the program does not proceed with the execution.

Consider the following example `enh_thr_mx_relock.c`

```
#include pthread.h
#include string.h
#include stdio.h
#include errno.h

pthread_mutex_t r_mtx;   /* recursive mutex */
pthread_mutex_t n_mtx;   /* normal mutex */
extern void     fatal_error(int err, char *func);
```

```
/* Print error information, exit with -1 status. */
void
fatal_error(int err_num, char *function)
{
        char    *err_string;

        err_string = strerror(err_num);
        fprintf(stderr, "%s error: %s\n", function, err_string);
        exit(-1);
}

#define check_error(return_val, msg) {          \
        if (return_val != 0)                            \
                fatal_error(return_val, msg);           \
}

main()
{
        pthread_mutexattr_t     mtx_attr;
        pthread_t               tid1;
        extern void             start_routine(int num);
        int                     ret_val;

        alarm (20);

        /* Initialize the mutex attributes */
        ret_val = pthread_mutexattr_init(&mtx_attr);
        check_error(ret_val, "mutexattr_init failed");

        /* Set the type attribute to recursive */
        ret_val = pthread_mutexattr_settype(&mtx_attr,
                        PTHREAD_MUTEX_RECURSIVE);
        check_error(ret_val, "mutexattr_settype failed");

        /* Initialize the recursive mutex */
        ret_val = pthread_mutex_init(&r_mtx, &mtx_attr);
        check_error(ret_val, "mutex_init failed");

        /* Set the type attribute to normal */
        ret_val = pthread_mutexattr_settype(&mtx_attr,
                        PTHREAD_MUTEX_NORMAL);
        check_error(ret_val, "mutexattr_settype failed");

        /* Initialize the normal mutex */
        ret_val = pthread_mutex_init(&n_mtx, &mtx_attr);
        check_error(ret_val, "mutex_init failed");

        /* Destroy the attributes object */
        ret_val = pthread_mutexattr_destroy(&mtx_attr);
        check_error(ret_val, "mutexattr_destroy failed");
```

```
        /* Rest of application code here */

        /*
         * Create a thread
         */
        ret_val = pthread_create(&tid1, (pthread_attr_t *)NULL,
                (void *(*)())start_routine, (void *)1);
        check_error(ret_val, "pthread_create 1 failed");

        /*
         * Wait for the threads to finish
         */
        ret_val = pthread_join(tid1, (void **)NULL);
        check_error(ret_val, "pthread_join: tid1");
}

void
start_routine(int thread_num)
{
        int     ret_val;

        sched_yield();

        /* Lock the recursive lock recursively. */
        ret_val = pthread_mutex_lock(&r_mtx);
        check_error(ret_val, "mutex_lock r_mtx");
        printf("Thread %d - got r_mtx\n", thread_num);

        ret_val = pthread_mutex_lock(&r_mtx);
        check_error(ret_val, "mutex_lock r_mtx");
        printf("Thread %d - got r_mtx\n", thread_num);
        ret_val = pthread_mutex_unlock(&r_mtx);
        check_error(ret_val, "mutex_unlock r_mtx");
        printf("Thread %d - released r_mtx\n", thread_num);

        ret_val = pthread_mutex_unlock(&r_mtx);
        check_error(ret_val, "mutex_unlock r_mtx");
        printf("Thread %d - released r_mtx\n", thread_num);

        /* Try locking the non-recursive lock recursively */
        ret_val = pthread_mutex_lock(&n_mtx);
        check_error(ret_val, "mutex_lock n_mtx");
        printf("Thread %d - got n_mtx\n", thread_num);

        ret_val = pthread_mutex_lock(&n_mtx);
        check_error(ret_val, "mutex_lock n_mtx");
        printf("Thread %d - got n_mtx\n", thread_num);

        ret_val = pthread_mutex_unlock(&n_mtx);
        check_error(ret_val, "mutex_unlock n_mtx");
        printf("Thread %d - released n_mtx\n", thread_num);
```

```
        ret_val = pthread_mutex_unlock(&n_mtx);
        check_error(ret_val, "mutex_unlock n_mtx");
        printf("Thread %d - released n_mtx\n", thread_num);
}
```

At run-time, the debugger keeps track of each mutex in the application and the thread that currently holds each mutex. When a thread attempts to acquire a lock on a non-recursive mutex, the debugger checks if the thread currently holds the lock object for the mutex.

```
(gdb) set thread-check recursive-relock on
```

The debugger transfers the execution control to the user and prints a warning message when this condition is detected.

The following is a segment of the HP WDB output:

```
Starting program: /home/gdb/enh_thr_mx_relock
Thread 1 - got r_mtx
Thread 1 - got r_mtx
Thread 1 - released r_mtx
Thread 1 - released r_mtx
Thread 1 - got n_mtx
[Switching to thread 2 (system thread 39774)]
warning: Attempt to recursively acquire non-recursive mutex 2 from thread 2.
```

TIP: Release the lock on a non-recursive mutex before attempting to acquire lock on the same object again, to avoid this situation.

Problem: The thread attempts to unlock a mutex or a read-write lock that it does not control.

Consider the following scenario: Thread 1 locks mutex A. Thread 2 unlocks mutex A. This is clearly an attempt from Thread 2 to release the lock on mutex A which was previously locked by Thread 1.

Consider the following example `enh_thr_unlock_not_own.c`:

```
#include pthread.h
#include string.h
#include stdio.h
#include errno.h
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

/* Print error information, exit with -1 status. */
void
fatal_error(int err_num, char *function)
{
        char    *err_string;

        err_string = strerror(err_num);
        fprintf(stderr, "%s error: %s\n", function, err_string);
        exit(-1);
}
```

```
#define check_error(return_val, msg) {          \
        if (return_val != 0)                            \
                fatal_error(return_val, msg);           \
}

main()
{
        pthread_t               tid1;
        extern void             start_routine(int num);
        int                     ret_val;

        /*
         * Create a thread
         */
        ret_val = pthread_create(&tid1, (pthread_attr_t *)NULL,
                (void *(*)())start_routine, (void *)1);
        check_error(ret_val, "pthread_create 1 failed");

        /*
         * Wait for the threads to finish
         */
        ret_val = pthread_join(tid1, (void **)NULL);
        check_error(ret_val, "pthread_join: tid1");
}

void
start_routine(int thread_num)
{
        int     ret_val;
        ret_val = pthread_mutex_unlock(&mtx);
        check_error(ret_val, "mutex_unlock mtx");
}
```

This usually is indicative of error in the program logic. Typically, applications are coded to lock and unlock objects on a one-one basis.

The following command enables you to detect this condition in a threaded application.

```
(gdb) set thread-check unlock_not_own on
```

The debugger transfers the execution control to the user and prints a warning message when this condition is detected.

The following is a segment of the HP WDB output:

```
Starting program: /home/gdb/enh_thr_unlock_not_own
[Switching to thread 2 (system thread 39941)]
warning: Attempt to unlock mutex 1 not owned by thread 2.
0x800003ffeffcc608 in __rtc_pthread_dummy+0 () from ../librtc64.sl
```

NOTE: In some rare predictable situations the thread might attempt to unlock an object that it has no control over. For example, an application which instructs the thread to unlock a mutex when it encounters a C++ destructor, irrespective of the history of the processing of the C++ constructor.

Problem: The Thread waits on a mutex or a read-write lock that is held by a thread with a different scheduling policy

Consider the following scenario:

Thread 1 is scheduled using Policy1, `SP1`. Thread 2 is scheduled using Policy2, `SP2`. Thread 1 waits for a read-write lock object which is held by Thread 2. Since the scheduling policy of the threads is not the same, there are chances of delay in Thread 2 releasing the lock for the read-write object.

Consider the following example `enh_thr_mixed_sched.c`:

```
#include pthread.h
#include errno.h
#include sched.h
#include stdio.h

extern void     *thread1_func(), *thread2_func();
extern void     fatal_error(int err_num, char *func);

pthread_mutex_t         mtx = PTHREAD_MUTEX_INITIALIZER;

/* Print error information, exit with -1 status. */
void
fatal_error(int err_num, char *function)
{
        char    *err_string;

        err_string = strerror(err_num);
        fprintf(stderr, "%s error: %s\n", function, err_string);
        exit(-1);
}

#define check_error(return_val, msg) {                  \
                if (return_val != 0)                    \
                        fatal_error(return_val, msg);   \
        }

void *
thread1_func()
{
        int     ret_val;

        ret_val = pthread_mutex_lock(&mtx);
        check_error(ret_val, "mutex_lock mtx");
        printf("In thread1_func()\n");
        sleep(5);
        ret_val = pthread_mutex_unlock(&mtx);
        check_error(ret_val, "mutex_unlock mtx");
```

```
        return((void *)NULL);
}

void *
thread2_func()
{
        int     ret_val;

        ret_val = pthread_mutex_lock(&mtx);
        check_error(ret_val, "mutex_lock mtx");
        printf("In thread2_func()\n");
        sleep(5);
        ret_val = pthread_mutex_unlock(&mtx);
        check_error(ret_val, "mutex_unlock mtx");

        return((void *)NULL);
}

main()
{
        pthread_t       pth_id[2];
        int             ret_val, scope;
        int             old_policy;
        pthread_attr_t  attr;
        struct sched_param      param, old_param;

        /* Initialize the threads attributes object */
        ret_val = pthread_attr_init(&attr);
        check_error(ret_val, "attr_init()");

        /* We want bound threads if they are available. */
        ret_val = pthread_attr_getscope(&attr, &scope);
        check_error(ret_val, "attr_getscope()");
        if (scope != PTHREAD_SCOPE_SYSTEM) {
                scope = PTHREAD_SCOPE_SYSTEM;
                ret_val = pthread_attr_setscope(&attr, scope);
                if ((ret_val != 0) && (ret_val != ENOTSUP))
                        fatal_error(ret_val, "attr_setscope()");
        }

        /* Thread 1 is a high priority SCHED_FIFO thread.*/
        ret_val = pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
        check_error(ret_val, "attr_setschedpolicy() 1");

        param.sched_priority = sched_get_priority_max(SCHED_FIFO);
        ret_val = pthread_attr_setschedparam(&attr, &param);
        check_error(ret_val, "attr_setschedparam() 1");

        ret_val = pthread_create(&pth_id[0], &attr, thread1_func, NULL);
        check_error(ret_val, "pthread_create() 1");

        /* Thread 2 is a low priority SCHED_RR thread. */
        ret_val = pthread_attr_setschedpolicy(&attr, SCHED_RR);
        check_error(ret_val, "attr_setschedpolicy() 2");

        param.sched_priority = sched_get_priority_min(SCHED_RR);
```

```
        ret_val = pthread_attr_setschedparam(&attr, &param);
        check_error(ret_val, "attr_setschedparam() 2");

        ret_val = pthread_create(&pth_id[1], &attr, thread2_func, NULL);
        check_error(ret_val, "pthread_create() 2");

        /* Destroy the thread attributes object */
        ret_val = pthread_attr_destroy(&attr);
        check_error(ret_val, "attr_destroy()");

        /* wait for the threads to finish */
        ret_val = pthread_join(pth_id[0], (void **)NULL);
        check_error(ret_val, "pthread_join() 1");

        ret_val = pthread_join(pth_id[1], (void **)NULL);
        check_error(ret_val, "pthread_join() 2");
}
```

Such a situation does not necessarily result in a deadlock or application errors. However, there might be instances of performance lag issues resulting from the mixed scheduling policies.

The following command enables you to check this condition in a threaded application.

```
set thread-check mixed-sched-policy[on|off]
```

The following is a segment of the HP WDB output:

```
Starting program: /home/gdb/enh_thr_mixed_sched
In thread1_func()
[Switching to thread 3 (system thread 39724)]
warning: Attempt to synchronize threads 3 and 2 with different scheduling policies.
0x800003ffeffcc608 in __rtc_pthread_dummy+0 () from ../librtc64.sl
```

TIP: Consider changing the application such that the threads with the same scheduling policy share the mutex.

Problem: Different threads non-concurrently wait on the same condition variable, but with different associated mutexes.

Consider the following scenario:

Thread 1 with mutex A waiting on conditional variable CV1.

Thread 2 with mutex B waiting on conditional variable CV1.

Consider the following example `enh_thr_cv_multiple_mxs.c`

```
#include pthread.h
#include stdlib.h
#include errno.h
pthread_mutex_t job_lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t job_lock2 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  job_cv = PTHREAD_COND_INITIALIZER;
extern void fatal_error(int err, char *f);

void
```

```
producer_thread(pthread_mutex_t* job_lock)
{
        int     ret_val;

        /* Acquire the associated mutex lock */
        if ((ret_val = pthread_mutex_lock(job_lock)) != 0)
          fatal_error(ret_val, "p mtx_lock failed");

        /* Signal the condvar to wakeup one thread */
        if ((ret_val = pthread_cond_signal(&job_cv)) != 0)
          fatal_error(ret_val, "cond_signal failed");

        /* Release the associated mutex */
       if ((ret_val = pthread_mutex_unlock(job_lock)) != 0)
          fatal_error(ret_val, "mtx_unlock failed");
}


void
consumer_thread(pthread_mutex_t* job_lock)
{
        int             ret_val;

        /* Acquire the condvar's associated mutex lock */
        if ((ret_val = pthread_mutex_lock(job_lock)) != 0)
          fatal_error(ret_val, "c mtx_lock failed");

        pthread_cond_wait(&job_cv, job_lock);

        /* Release the associated mutex */
        if ((ret_val = pthread_mutex_unlock(job_lock)) != 0)
          fatal_error(ret_val, "mtx_unlock failed");

}


#define check_error(return_val, msg) {                  \
                if (return_val != 0)                    \
                        fatal_error(return_val, msg);   \
        }

int main(int argc, char* argv[])
{
        pthread_t       tid1, tid2, tid3, tid4;
        pthread_mutex_t *l1, *l2;
        int             ret_val;

        if (argc == 1) {
          fprintf(stderr, "error: no arguments\n");
          exit (1);
        }
        else if (strcmp(argv[1], "bad") == 0) {
```

```
          l1 = &job_lock1
          l2 = &job_lock2
        }
        else {
          l1 = l2 = &job_lock1
 }

        alarm(20);

        /* Create two threads to do the work */
        ret_val = pthread_create(&tid1, (pthread_attr_t *)NULL,
                (void *(*)())consumer_thread, (void *) l1);
        check_error(ret_val, "pthread_create 1 failed");

        ret_val = pthread_create(&tid2, (pthread_attr_t *)NULL,
                (void *(*)())producer_thread, (void *) l1);
        check_error(ret_val, "pthread_create 2 failed");

        if (l1 != l2) {
          ret_val = pthread_create(&tid3, (pthread_attr_t *)NULL,
                    (void *(*)())consumer_thread, (void *) l2);
          check_error(ret_val, "pthread_create 1 failed");

          ret_val = pthread_create(&tid4, (pthread_attr_t *)NULL,
                  (void *(*)())producer_thread, (void *) l2);
          check_error(ret_val, "pthread_create 2 failed");
        }
      /* Wait for the threads to finish */
        ret_val = pthread_join(tid1, (void **)NULL);
        check_error(ret_val, "pthread_join: tid1");

        ret_val = pthread_join(tid2, (void **)NULL);
        check_error(ret_val, "pthread_join: tid2");

        if (l1 != l2) {
          ret_val = pthread_join(tid3, (void **)NULL);
          check_error(ret_val, "pthread_join: tid3");

          ret_val = pthread_join(tid4, (void **)NULL);
          check_error(ret_val, "pthread_join: tid4");
        }

        exit(0);
}

void
fatal_error(int err_num, char *function)
{
        char    *err_string;
  err_string = strerror(err_num);
        fprintf(stderr, "%s error: %s\n", function, err_string);
        exit(-1);
```

```
}
```

The following command enables you to check this condition in a threaded application.

```
(gdb) set thread-check cv-multiple-mxs[on|off]
```

The debugger transfers the execution control to the user and prints a warning message when this condition is detected.

The following is a segment of the HP WDB output:

```
Starting program: /home/gdb/enh_thr_cv_multiple_mxs bad
[Switching to thread 4 (system thread 39531)]
warning: Attempt to associate condition variable 0 with mutexes 1 and 2.
0x800003ffeffcc608 in __rtc_pthread_dummy+0 () from ../librtc64.sl
```

According to pthread implementation, the threads that concurrently wait on a single conditional variable need to specify the same associated mutex.

TIP: The solution is to correct the application source code in such a way that the condition variable which violates the rule uses the same mutex.

Problem: The thread terminates execution without unlocking the associated mutexes or read-write locks.

Consider the following scenario:

Thread A holds mutex `MX`. This thread terminates without unlocking the mutex `MX`. There are other threads in the program that wait to gain control on `MX`.

The termination of thread A with the locked mutex `MX`causes the other threads to be in an endless wait to gain control on `MX`. This situation is an example of a deadlock where the program execution is dependent on the locked mutex and hence is unable to proceed.

Consider the following example `enh_thr_exit_own_mx.c`

```
#include pthread.h
#include stdlib.h
#include errno.h
pthread_mutex_t job_lock1 = PTHREAD_MUTEX_INITIALIZER;
extern void  fatal_error(int err, char *f);

void
producer_thread(pthread_mutex_t* job_lock)
{
        int     ret_val;

        /* Acquire the associated mutex lock */
        if ((ret_val = pthread_mutex_lock(job_lock)) != 0)
          fatal_error(ret_val, "p mtx_lock failed");
}

#define check_error(return_val, msg) {                  \
```

```
                if (return_val != 0)                    \
                        fatal_error(return_val, msg);   \
        }

main()
{
        pthread_t       tid;
        int             ret_val;

        /* Create two threads to do the work */
        ret_val = pthread_create(&tid, (pthread_attr_t *)NULL,
        (void *(*)())producer_thread, (void *) &job_lock1);
        check_error(ret_val, "pthread_create 2 failed");

        /* Wait for the threads to finish */
        ret_val = pthread_join(tid, (void **)NULL);
        check_error(ret_val, "pthread_join: tid");

        exit(0);
}

void
fatal_error(int err_num, char *function)
{
        char    *err_string;

        err_string = strerror(err_num);
        fprintf(stderr, "%s error: %s\n", function, err_string);
        exit(-1);
}
```

The following command enables you to check this condition in a threaded application:

```
set thread-check thread-exit-own-mutex [on|off]
```

In such a scenario, the debugger transfers the execution control to the user and displays a warning message.

The following is a segment of the HP WDB output:

```
Starting program: /home/gdb/enh_thr_exit_own_mx
[Switching to thread 2 (system thread 39677)]
warning: Attempt to exit thread 2 while holding a mutex 1.
0x800003ffeffcc608 in __rtc_pthread_dummy+0 () from ../librtc64.sl
```

TIP:

- If the remaining segments of the application require access to the locked mutex, modify the code segment of the terminating thread to unlock the mutex before it terminates.
- If the termination is the result of an exception, then consider using a condition handler (in C++) or POSIX Threads library TRY/FINALLY blocks.

Problem: The thread waits on a condition variable for which the associated mutex is not locked.

Consider the following scenario:

A function has a thread which is associated to the mutex MX. The function calls the POSIX Thread Library routine pthread_cond_wait() beforeMX is locked.

Consider the following example enh_thr_cv_wait_no_mx.c:

```
#include pthread.h
#include stdlib.h
#include errno.h

pthread_mutex_t job_lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t job_lock2 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  job_cv = PTHREAD_COND_INITIALIZER;
extern void  fatal_error(int err, char *f);

void
producer_thread(pthread_mutex_t* job_lock)
{
        int     ret_val;

        /* Acquire the associated mutex lock */
        if ((ret_val = pthread_mutex_lock(job_lock)) != 0)
          fatal_error(ret_val, "p mtx_lock failed");

        /* Signal the condvar to wakeup one thread */
        if ((ret_val = pthread_cond_signal(&job_cv)) != 0)
          fatal_error(ret_val, "cond_signal failed");

        /* Release the associated mutex */
       if ((ret_val = pthread_mutex_unlock(job_lock)) != 0)
          fatal_error(ret_val, "mtx_unlock failed");
}

void
consumer_thread(pthread_mutex_t* job_lock)
{
        int ret_val;
        pthread_cond_wait(&job_cv, job_lock);
}
```

```
#define check_error(return_val, msg) {                  \
                if (return_val != 0)                    \
                        fatal_error(return_val, msg);   \
        }

main()
{
        pthread_t tid1, tid2;
        int ret_val;

        alarm (20);

        /* Create two threads to do the work */
        ret_val = pthread_create(&tid1, (pthread_attr_t *)NULL,
        (void *(*)())consumer_thread, (void *) &job_lock1);
        check_error(ret_val, "pthread_create 1 failed");

        ret_val = pthread_create(&tid2, (pthread_attr_t *)NULL,
        (void *(*)())producer_thread, (void *) &job_lock1);
        check_error(ret_val, "pthread_create 2 failed");

        /* Wait for the threads to finish */
        ret_val = pthread_join(tid1, (void **)NULL);
        check_error(ret_val, "pthread_join: tid1");

        ret_val = pthread_join(tid2, (void **)NULL);
        check_error(ret_val, "pthread_join: tid2");

        exit(0);
}

void
fatal_error(int err_num, char *function)
{
        char    *err_string;

        err_string = strerror(err_num);
        fprintf(stderr, "%s error: %s\n", function, err_string);
        exit(-1);
}
```

This scenario, where a thread waits on a conditional variable before the associated mutex is locked, is a potential cause of unpredictable results in `POSIX`library.

The following command in HP WDB enables you to check this condition in a threaded application:

```
set thread-check cv-wait-no-mx[on|off]
```

In such a scenario, the debugger transfers the execution control to the user and displays a warning message.

The following is a segment of the HP WDB output:

```
Starting program: /home/gdb/enh_thr_cv_wait_no_mx
[Switching to thread 2 (system thread 39559)]
warning: Attempt by thread 2 to wait on condition variable 0 without locking the associated mutex
1.
0x800003ffeffcc608 in __rtc_pthread_dummy+0 () from ../librtc64.sl
```

NOTE: This is an additional check that HP WDB provides and is not a `POSIX.1` standard requirement for the `pthread_cond_wait()` routine.

You can determine the function which attempts to wait on the condition, by looking at the batcktrace(`bt`) of the thread that is reported above.

TIP: Modify the code segment of the function so it acquires control over the mutex associated to the conditional variable before it calls the routine, `pthread_cond_wait()`.

Problem: The thread terminates execution, and the resources associated with the thread continue to exist in the application because the thread has not been joined or detached.

Consider the following scenario:

Thread A in an application terminates execution successfully or as a result of an exception/cancel. The resources associated with the thread exist in the application until the thread is joined or detached.

Consider the following example `enh_thr_exit_no_join_detach.c`:

```
#include pthread.h
#include stdlib.h
#include errno.h

pthread_mutex_t job_lock1 = PTHREAD_MUTEX_INITIALIZER;
extern void fatal_error(int err, char *f);

void
my_thread(void* num)
{
        int ret_val;

        /* Acquire the associated mutex lock */
        if ((ret_val = pthread_mutex_lock(&job_lock1)) != 0)
          fatal_error(ret_val, "p mtx_lock failed");

        printf ("In thread %d\n", (int) num);

        /* Release the associated mutex */
        if ((ret_val = pthread_mutex_unlock(&job_lock1)) != 0)
          fatal_error(ret_val, "mtx_unlock failed");
}

#define check_error(return_val, msg) {                  \
                if (return_val != 0)                    \
```

```
                        fatal_error(return_val, msg);   \
        }

main()
{
        pthread_t       tid1, tid2, tid3;
        int             ret_val;

        /* Create two threads to do the work */
        ret_val = pthread_create(&tid1, (pthread_attr_t *)NULL,
                (void *(*)())my_thread, (void *)1);
        check_error(ret_val, "pthread_create 1 failed");

        ret_val = pthread_create(&tid2, (pthread_attr_t *)NULL,
                (void *(*)())my_thread, (void *)2);
        check_error(ret_val, "pthread_create 2 failed");

        ret_val = pthread_create(&tid3, (pthread_attr_t *)NULL,
                (void *(*)())my_thread, (void *)3);
        check_error(ret_val, "pthread_create 3 failed");
   /* Detach thread 1 */
        ret_val = pthread_detach(tid1);
        check_error(ret_val, "pthread_join: tid");

        sleep(5);

        /* Wait for the thread 2 to finishes */
        ret_val = pthread_join(tid2, (void **)NULL);
        check_error(ret_val, "pthread_join: tid");

        exit(0);
}

void
fatal_error(int err_num, char *function)
{
        char    *err_string;

        err_string = strerror(err_num);
        fprintf(stderr, "%s error: %s\n", function, err_string);
        exit(-1);
}
```

If an application repeatedly created threads without the join or detach operation, it will leak resources that might eventually cause the application to fail.

The following command enables you to check this condition in a threaded application:

```
set thread-check thread-exit-no-join-detach[on|off]
```

In such a scenario, the debugger transfers the execution control to the user and displays a warning message.

The following is a segment of the HP WDB output:

```
Starting program: /home/gdb/enh_thr_exit_no_join_detach
In thread 1
In thread 2
In thread 3
warning: Attempt to exit thread 4 which has neither been joined nor detached.
0x800003ffeffcc608 in __rtc_pthread_dummy+0 () from ../librtc64.sl
```

NOTE: A violation of this condition implies outstanding resources that are not released. If the number of violations is small, or if they occur on an error path that causes abrupt termination of the application, you can disable this check on threads.

TIP:

- You can modify the application so that the thread is joined or detached.
- If a thread is never cancelled, joined, or otherwise passed to threads API, it needs to be detached. The thread can be explicitly joined or detached or created with the detach attribute.
- If the thread is to be manipulated by a threads API, or the return value of threads is to be considered, then the join operation should be performed to obtain the value and destroy the thread.

Problem: The thread uses more than the specified percentage of the stack allocated to the thread.

Each thread is assigned a specific percentage of the stack when it is created. If the stack allocation is not specified for a thread, the default value is used. The stack allocation cannot be modified after a thread is created.

The application must ensure that the thread stack size is sufficient for all operations of the thread. If a thread attempts to use more space than the allocated stack space, it results in a stack overflow.

Consider the following example:

```
#include pthread.h
#include stdlib.h
#include stdio.h
#include errno.h
pthread_mutex_t job_lock = PTHREAD_MUTEX_INITIALIZER;
extern void fatal_error(int err, char *f);

void
my_thread()
{
        int     ret_val;
        int more_stack[100];
        static int count = 0;

        sched_yield();

        /* Acquire the associated mutex lock */
```

```
        if ((ret_val = pthread_mutex_lock(&job_lock)) != 0)
          fatal_error(ret_val, "p mtx_lock failed");

        for (int i = 0; i < 100; i++)
          more_stack[i] = i;
        for (int i = 0; i < 1000; i++);

        /* Release the associated mutex */
        if ((ret_val = pthread_mutex_unlock(&job_lock)) != 0)
          fatal_error(ret_val, "mtx_unlock failed");

        my_thread();
}


#define check_error(return_val, msg) {                  \
                if (return_val != 0)                    \
                        fatal_error(return_val, msg);   \
        }

main()
{
        pthread_t       tid;
        int             ret_val;

        /* Create two threads to do the work */
        ret_val = pthread_create(&tid, (pthread_attr_t *)NULL,
                (void *(*)())my_thread, (void *) NULL);
  check_error(ret_val, "pthread_create 2 failed");

        /* Wait for the threads to finish */
        ret_val = pthread_join(tid, (void **)NULL);
        check_error(ret_val, "pthread_join: tid");

        exit(0);
}

void
fatal_error(int err_num, char *function)
{
        char    *err_string;

        err_string = strerror(err_num);
        fprintf(stderr, "%s error: %s\n", function, err_string);
        exit(-1);
}
```

The `set thread-check stack-util[num]` command checks if any thread has used more than the specified percentage[`num`] of the stack allocation.

The debugger transfers the execution control to the user and displays a warning message when this condition is detected.

The following is a segment of the HP WDB output:

```
(gdb) set thread-check stack-util 101
Invalid value: stack utilization must be between 0 and 100.
(gdb) set thread-check stack-util 80
(gdb) run
Starting program: /home/gdb/enh_thr_stack_util
[Switching to thread 2 (system thread 39877)]
warning: Thread 2 exceeded stack utilization threshold of 80%.
0x800003ffeffcc608 in __rtc_pthread_dummy+0 () from ../librtc64.sl
```

This warning indicates that the thread attempts to exceed its stack utilization limit. This may cause memory access violations, bus errors, or segmentation faults, if the stack utilization reaches 100%.

TIP:

- Increase the stack allocation available to the thread when you create it
- Change the code running in the thread to reduce its use of stack space

Problem: The number of threads waiting on any pthread object exceeds the specified threshold number.

This check identifies contention that results from too many threads attempting to acquire the same lock object.

The `set thread-check num-waiters [num]` command checks if the number of threads waiting on any `pthread` object exceeds the specified threshold number [`num`].

The debugger transfers the execution control to the user and displays a warning message when this condition is detected.

A relatively large number of threads waiting on pthread synchronization object can indicate a performance constraint on the application.

TIP: To avoid this condition:

- Check if any of the data object shared among the application threads can be accessed using its own synchronization object.
- Check if the program has too many threads whose activity depends on concurrent access to the contended mutex.

Thread-debugging in Batch mode

HP WDB 5.8 supports batch mode of debugging threads for HP-UX 11iv2 and later, on Integrity systems and on HP-UX 11i v3 in PA-RISC systems for 64 bit applications.

The debugger provides a log file with the list of thread-related errors that occur in the application.

In batch mode, the debugger detects the all the thread-conditions that are detected during an interactive debugging session.

The debugger reports extended information such as variable address, name, id and other specifications related to the involved `pthread` objects. In addition, it displays the stack trace of the executing thread at the point of error.

Pre-requisites for Batch Mode of Thread Debugging

The various prerequisites for Batch mode of Thread Debugging are as follows:

- The thread-debugging feature in HP WDB is dependent on the availability of the dynamic linker B.11.19 or later versions.
- Advanced thread-debugging feature requires the pthread tracer library which is available by default on systems running HP-UX 11i v2 or later.

Steps to debug threads in Batch Mode

1. Compile the source files.

   Set the `LD_LIBRARY_PATH` environment variable, based on the platform as follows:

   - For IPF 32 bit applications, set

     ```
     export LD_LIBRARY_PATH=/opt/langtools/wdb/lib/hpux32
     ```

   - For IPF 64 bit applications, set

     ```
     export LD_LIBRARY_PATH=/opt/langtools/wdb/lib/hpux64
     ```

   - For PA 64 bit applications, set

     ```
     export LD_LIBRARY_PATH=/opt/langtools/wdb/lib/pa20_64
     ```

2. Map the share libraries as private for HP 9000 systems using the following command:

   ```
   $ chatr +dbg enable ./executable
   ```

NOTE: This step is not applicable for Integrity systems.

3. Create a configuration file, `rtcconfig` to specify the various thread conditions that you want the debugger to detect.

NOTE: The configuration file contains lines of the following form:

```
set thread-check [on|off] | [option] [on|off] | [option] [num]
```

And/Or

```
set frame-count [num]
```

And/Or

```
files = <name of the executable on which the thread checking is to be done>
```

4. Set the environment variable `BATCH_RTC` to on as `export set BATCH_RTC=on`
5. Complete one of the following steps to preload the `librtc` runtime library:
   - Set the target application to preload **librtc** by using the `+rtc` option for the `chatr` command. In addition to automatically loading the `librtc` library, the `+rtc` option for the `chatr` command also maps the shared libraries as private. To enable or disable the target application to preload the `librtc` runtime library, enter the following command at the HP-UX prompt:

     ```
     $ chatr +rtc <enable|disable> <executable>
     ```

     NOTE: The `chatr +rtc` option preloads the `librtc` runtime library from the following default paths:
     — For 32 bit IPF applications,

     ```
     /opt/langtools/lib/hpux32/librtc.so
     ```

     — For 64 bit IPF applications,

     ```
     /opt/langtools/lib/hpux64/librtc.so
     ```

     — For 64-bit PA applications,

     ```
     /opt/langtools/lib/pa20_64/librtc.sl
     ```

     To preload the `librtc` runtime library from a path that is different from the default paths, you must use the `LD_PRELOAD` environment variable.
   - Instead of automatically preloading `librtc` and mapping the shared libraries, you can explicitly preload the required `librtc` library after mapping the shared libraries private.

     In the case of HP 9000 systems, you must explicitly map the share libraries as private by using the `+dbg` enable option for the `chatr` command, as follows:

     ```
     $ chatr +dbg enable ./<executable>
     ```

     (This step is not required on Integrity systems.)

To explicitly preload the `librtc` runtime library and start the target application, enter one of the following commands:

— For 32 bit IPF applications,

```
LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so
<executable>
```

— For 64 bit IPF applications,

```
LD_PRELOAD=/opt/langtools/lib/hpux64/librtc.so
<executable>
```

— For 64-bit PA applications,

```
LD_PRELOAD=/opt/langtools/lib/pa20_64/librtc.sl
<executable>
```

If `LD_PRELOAD` and `chatr +rtc` are used to preload the `librtc` runtime library, the `librtc` runtime library is loaded from the path specified by `LD_PRELOAD`.

If HP WDB detects any thread error condition during the application run, the error log is output to a file in the current working directory.

The output file has the following naming convention:

```
<executablename>.<pid>.threads
```

where `pid` is the process id.

Limitations in Batch Mode of Thread Debugging

The feature does not obtain the thread-error information in batch mode for forked process in a multiprocessing application. However, if the `librtc.sl` library is preloaded, the debugger obtains the thread-error information in the batch mode for `exec`-ed application.

You cannot specify an alternate output directory for the thread-error log. The thread-error log file is output into the current working directory only.

HP WDB cannot execute both batch mode thread check and batch mode heap check together. If the rtcconfig file has both entries, then batch heap check overrides the batch thread check.

Known issues with thread debugging for interactive and batch mode

Issue 1:

During the execution of advanced thread checking for applications that fork, in the interactive mode, the following message appears if the GDB follows the child:

```
Pthread analysis file missing!
```

This error message appears because the thread-error information for the forked process is not available.

However, if the forked process `exec()`s another binary, the thread-error information is available for the `exec` -ed binary.

Issue 2:

In both interactive and batch modes, if the applications exceed their thread stack utilization, the following error message appears:

```
Error accessing memory address
```

This occurs when GDB attempts a command line call on an already overflowing thread stack.

Thread- debugging in Attach Mode

HP WDB provides support to attach a running process to the debugger. To use thread debugging commands after attaching GDB to a running process, complete the following steps:

1. Set `LD_LIBRARY_PATH` to include the appropriate directory, by entering one of the following commands:
   - For 32 bit IPF applications,

     ```
     export LD_LIBRARY_PATH=/opt/langtools/wdb/lib/hpux32
     ```
   - For 64 bit IPF applications,

     ```
     export LD_LIBRARY_PATH=/opt/langtools/wdb/lib/hpux64
     ```
   - For 32 bit PA applications,

     ```
     export LD_LIBRARY_PATH=/opt/langtools/wdb/lib
     ```
   - For 64-bit PA applications,

     ```
     export LD_LIBRARY_PATH=/opt/langtools/wdb/lib/pa20_64
     ```
2. Complete one of the following steps to preload the `librtc` runtime library:
   - Set the target application to preload `librtc` by using the `+rtc` option for the `chatr` command. In addition to automatically loading the `librtc` library, the `+rtc` option for the `chatr` command also maps the shared libraries as private.

     To enable or disable the target application to preload the `librtc` runtime library, enter the following command at the HP-UX prompt:

     ```
     $ chatr +rtc <enable|disable> <executable>
     ```

NOTE: The `chatr +rtc` option preloads the `librtc` runtime library from the following default paths:

— For 32-bit IPF applications,

```
/opt/langtools/lib/hpux32/librtc.so
```

— For 64-bit IPF applications,

```
/opt/langtools/lib/hpux64/librtc.so
```

— For 32-bit PA applications,

```
opt/langtools/lib/librtc.sl
```

— For 64-bit PA applications,

```
/opt/langtools/lib/pa20_64/librtc.sl
```

To preload the `librtc` runtime library from a path that is different from the default paths, you must use the `LD_PRELOAD` environment variable.

- Instead of automatically preloading `librtc` and mapping the shared libraries, you can explicitly preload the required `librtc` library after mapping the shared libraries private.

  In the case of HP 9000 systems, you must explicitly map the share libraries as private by using the `+dbg` enable option for the `chatr` command, as follows:

  ```
  $ chatr +dbg enable ./<executable>
  ```

  (This step is not required on Integrity systems.)

  To explicitly preload the `librtc` runtime library and start the target application, enter one of the following commands:

  — For 32-bit IPF applications,

  ```
  LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so
  <executable>
  ```

  — For 64-bit IPF applications,

  ```
  LD_PRELOAD=/opt/langtools/lib/hpux64/librtc.so
  <executable>
  ```

  — For 32-bit PA applications,

  ```
  LD_PRELOAD=/opt/langtools/lib/librtc.sl <executable>
  ```

  — For 64-bit PA applications,

  ```
  LD_PRELOAD=/opt/langtools/lib/pa20_64/librtc.sl
  <executable>
  ```

If LD_PRELOAD and chatr +rtc are used to preload the librtc runtime library, the librtc runtime library is loaded from the path specified by LD_PRELOAD.

3. Complete one of the following steps:
   - Attach the debugger to the required process and enable thread debugging, as follows:

     ```
     gdb -thread -p <pid>
     ```

     or

     ```
     gdb -thread <executable> <pid>
     ```

   - Alternately, you can attach the process to the debugger and consequently invoke thread debugging, as follows:

     ```
     $ gdb <executable> <pid>
     ...
     (gdb)set thread-check on
     ```

Thread Debugging in +check Mode

The +check=thread compiler option enables batch mode thread debugging features of HP WDB.

NOTE: This feature is available only for compiler versions A.06.20 and later.

It is a convenient way of launching the batch mode advanced thread checking features without setting any other environment variables at runtime. In other words, batch mode thread checking has two modes of invocation. The first method is to use the run-time environment variables LD_LIBRARY_PATH, LD_PRELOAD and BATCH_RTC on existing precompiled applications. The second method is to use the +check=thread option at the compile time.

+check=thread must only be used with multithreaded programs. It is not enabled by +check=all. This functionality requires HP WDB 5.9 or later.

The default configuration used by +check=thread option is as follows:

```
thread-check=1;recursive-relock=1;unlock-not-own=1;
mix-sched-policy=1;cv-multiple-mxs=1;cv-wait-no-mx=1;
thread-exit-own-mutex=1;thread-exit-no-join-detach=1;stack-util=80;
num-waiters=0;frame_count=4;output_dir=.;
```

Behavior of the +check=thread option can be changed by users by providing their own rtcconfig file. The user specified rtcconfig file can be in the current directory or in a directory specified by the GDBRTC CONFIG environment variable.

If any thread error condition is detected during the application run, the error log will be output to a file in the current working directory. The output file will have the following naming convention:

```
<executable name>.<pid>.threads,
```

where, <pid> is the process identifier.

Miscellaneous notes on Advanced thread debugging feature in HP WDB

The following commands enable you to view extended information on threads, mutexes, read-write locks and conditional variables in multi-threaded applications:

`info thread [thread-id]`

The `info thread [thread-id]` command displays a list of known threads. If you provide a `thread-id`, the command displays extended information on the specified thread.

Consider the following example:

```
(gdb) info thread
system thread 4189 0x7f666da8
in __pthread_create_system+0x3d8 () from /usr/lib/libpthread.1
2 system thread 4188 worker (wptr=0x40004640 " ) at quicksort.c:135
1 system thread 4184 0x7f66f728 in _lwp_create+0x10 () from /usr/lib/libpthread.1
```

`info mutex [mutex-id]`

The `info mutex [mutex-id]` command displays a list of known mutexes. If a `mutex-id` is provided, the command displays extended information on the specified mutex.

`info condvar [condvar-id]`

The `info condvar [condvar-id]` command displays a list of known condition variables. If `condvar-id` is provided, the command displays extended information on the specified condition variable.

`info rwlock [rwlock-id]`

The `info rwlock [rwlock-id]` command displays a list of known read-write locks. If `rwlock-id` is provided, the command displays extended information on the specified read-write lock.

Best Practices in Thread programming

The following are some of the best practices specific to Thread-programming:

- Ensure your program locks small segments or specific fields of a program segment to increase concurrency in execution.
- Ensure that your locks always obtain synchronization control in an order that does not cause race condition.
- Minimize locking instances so you can reduce the overhead that might result from frequent synchronization efforts.
- Minimize critical sections that might result in longer waits for other threads.
- Reduce the number of nested function calls so you avoid errors resulting from stack overflow.

- Consider writing a recursive function in an iterative form, as sometimes an iterative function demonstrates greater resource efficiency than a recursive function.
- Minimize the size and number of the stack local variables to reduce the stack usage of a thread.
- Allocate large data items dynamically in a heap than in arrays to ensure that stack utilization does not exceed the allocated or default percentage.

Appendix

Pertinent terms used throughout the white paper.

Backtrace Backtrace is a summary of proceeds of execution of the program which displays the stack frame number and the function name output of the `bt` command.

Concurrency Concurrency is a concept where more than one thread performs different functions simultaneously to complete the execution of the process at the earliest.

Core file Core file consists of information such as memory image of the address space of specific processes, values of the process registers and the states of a specific program. This file is the result of an abnormally terminated program.

Debugging Debugging is to understand, identify, and fix errors in a program.

Heap Heap is a large pool of memory from which dynamic memory allocations are done for program elements such as large arrays, structures, or classes.

Multi-threaded application

Multi threaded applications consist of multiple threads sharing resources with each other.

Parallelism Parallelism is a concept where more than one thread performs the same function simultaneously to complete the execution of the process at the earliest.

Process Process is an instance of a program in execution. It provides an environment for the execution of the threads within. The environment consists of details such as process identification details, working directory, an address space and system resources such as file descriptors, signal actions, shared libraries, and shared memory.

Pthreads `POSIX` is a standard that defines Application Programming Interface to create and manipulate threads. `POSIX` Threads is a `POSIX` standard for Threads. Libraries that implement `POSIX` threads are named Pthreads.

Segmentation faults When a program attempts to access a memory location in an improper way, the error condition is called Segmentation fault. In addition, attempts to access a memory location to which the access is denied also results in segmentation fault.

Shared memory Shared memory is the segment of memory that is accessed by multiple programs to establish quick communication among them.

Stack Stack is an abstract data structure that stores temporary information in sequential set of memory addresses.

Stack frames Stack frames are machine dependent data structures in call stack that have data pertaining to the execution state of a function.

Stack overflows Stack overflow is a condition where more than the determined amount of memory is used on the call stack. This results in program crash.

Stack trace Stack trace is a summary of the stack frames. These stack frames contain the function calls which the program initiates during its execution.

Thread Debugging commands at a glance

Table 2 Thread Debugging Commands

Command	Description
`set thread-check on/off`	Enables or disables Advanced thread debugging feature. The default setting is `off`.
`info-thread[thread-id]`	Displays a list of known threads.
`info mutex[mutex-id]`	Displays a list of known mutexes.
`info condvar[condvar-id]`	Displays a list of known conditional variables.
`info rwlock[rwlock-id]`	Displays a list of known read-write locks.
`set thread-check recursive-relock[on|off]`	Checks if a thread attempts to acquire a non-recursive mutex that it currently holds.
`set thread-check stack-util[num]`	Checks if any thread has used more than the specified percentage [`num`] of the stack location.
`set thread-check num-waiters[num]`	Checks if the number of threads waiting on any pthread objects exceeds the specified threshold number.
`set thread-check thread-exit-no-join-detach[on|off]`	Checks if a thread has terminated execution without joining or detaching the thread.
`set thread-check unlock-not-own[on|off]`	Checks if a thread has attempted to unlock a mutex or a read-write block that it has not acquired.
`set thread-check mixed-sched-policy[on|off]`	Checks if a thread waits on a mutex or a read-write lock that is held by a thread with a different scheduling policy.
`set thread-check cv-multiple-mxs[on|off]`	Checks if an application uses the same condition variable in multiple calls by different threads.
`set thread-check cv-wait-no-mx[on|off]`	Checks if the associated mutex of a condition variable is locked when the thread calls the `pthread_cond_wait()` routine.