Debugging Common Issues in Multithreaded Applications

A Comparison of Debugging a Multithreaded Application using GNU GDB and TotalView

A White Paper by Rogue Wave Software.

Rogue Wave Software
5500 Flatiron Parkway,
Suite 200
Boulder, CO 80301, USA
www.roguewave.com

Debugging Common Issues in Multithreaded Applications

A Comparison of Debugging a Multithreaded Application using GDB and TotalView

by Rogue Wave Software

TABLE OF CONTENTS

Abstract

This paper describes several challenges that are commonly encountered when debugging multithreaded applications in order to compare the open source GNU GDB debugger (hereafter referred to as GDB) with the TotalView parallel debugger from Rogue Wave Software. The challenges highlighted will include:

- debugging a multithreaded application
- investigating nondeterministic issues
- demonstrating thread control capabilities
- displaying local variables

TotalView offers impressive feature capabilities that are either not included or not possible to perform with GDB. This paper explores some of these features and scenarios to demonstrate the differences between the two debugging tools.

Sample Multithreaded Application with Common Issues

In the examples found in this paper – we will be referring to an application. The actual code is not included in this discussion, but the following description outlines the important details of the application and the bugs we will be tracking down in the following scenarios.

The application is a simple tcp-server that creates a new thread (super-thread) for processing each incoming client request. Each super-thread spawns a set of sub-threads which approximate the value of pi using the trapezoidal rule. The server allows one to define the maximum number of super-threads and the number of sub-threads. Three implemented bugs can be switched on and off for demonstration purposes. This could be done through the use of global variables, or the use of ifdefs in the code and rebuilding the application for each scenario.

The application is implemented in C and consists of these four functions:

```
int main( int argc, char* argv[]) /* All super-threads are created here */

void *processRequest( void *sfd ) /* The sub-threads will be spawned in
                                     this function */

void *iterate( void *ind ) /* Sub-threads are calculating PI */


void onError( const char *errorMsg) /* Error handling */
```

Properties referring to each super-thread are stored in an array of structures which is globally visible. Whether a thread is still alive or has already been terminated, these structures will hold the thread id, the session id, and the results calculated by the super-thread's sub-threads.

Introducing such an array (threadInfo tID[MAXTHREADS];) allows one to provide thread functions with a pointer to an index of this array while creating a thread in order to hand over all relevant super-thread related properties.

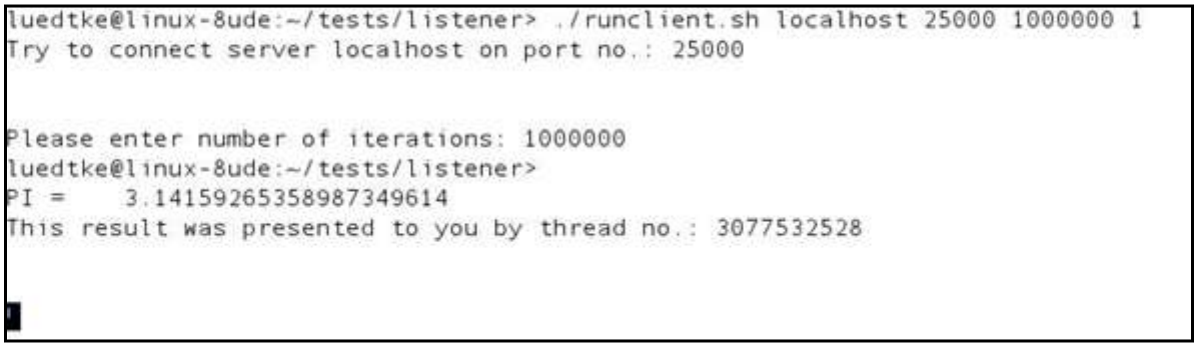
The following variables, which are used to store the index's integer value, will be mentioned in this document:

```
int *pindex; /* Pointer to an index of type integer, used in function
                main() as argument in pthread_create() */

void *sfd;   /* Void pointer to an index of type integer, used in function
                processRequest() */

int index;   /* Local integer variable used in function processRequest() */

void *ind;   /* Void pointer to an index of type integer, used in function
                iterate() */

int index;   /* Local integer variable used in function iterate() */
```

Debugging a Simple, Multithreaded Application

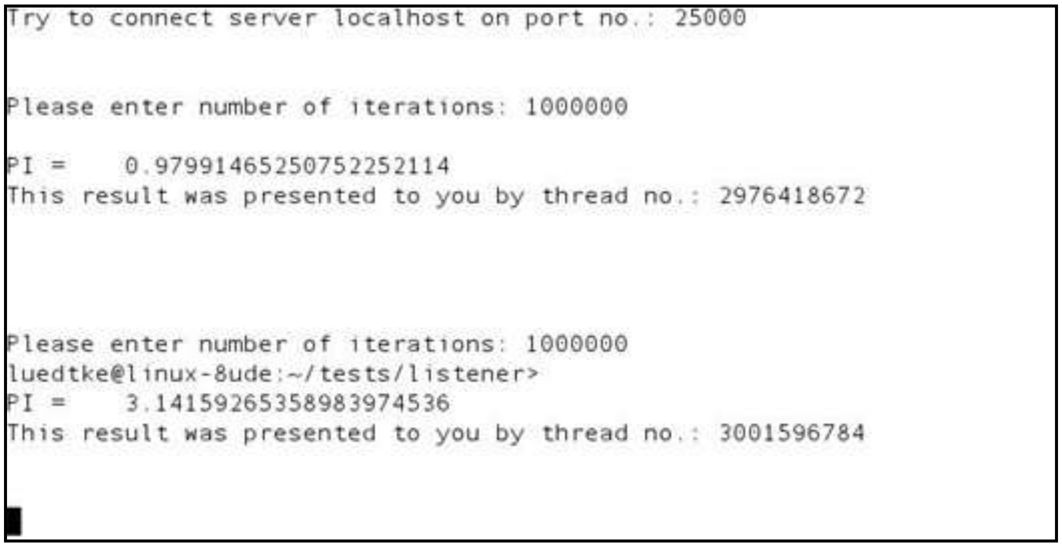
The following example compares the capabilities of GDB with TotalView when debugging a simple, multithreaded application.

The first bug should be easy to find. The server calculates reasonable results when only one super-thread is running, as seen in Figure 1. However, when additional super-threads are created the results become incorrect, as shown in Figure 2.



```
luedtke@linux-8ude:~/tests/listener> ./runclient.sh localhost 25000 1000000 1
Try to connect server localhost on port no.: 25000


Please enter number of iterations: 1000000
luedtke@linux-8ude:~/tests/listener>
PI =    3.14159265358987349614
This result was presented to you by thread no.: 3077532528
```

Figure 1: Correct results when one super-thread is run



```
Try to connect server localhost on port no.: 25000


Please enter number of iterations: 1000000

PI =    0.97991465250752252114
This result was presented to you by thread no.: 2976418672




Please enter number of iterations: 1000000
luedtke@linux-8ude:~/tests/listener>
PI =    3.14159265358983974536
This result was presented to you by thread no.: 3001596784
```

Figure 2: Incorrect results when two super-threads are running

It seems that there is a conflict between threads that occurs in cases where more than one super-thread is running. The popular (some would say ubiquitous) GDB debugger[1] will be used to demonstrate this issue. We will also show how that same issue can be addressed using TotalView[2].

Debugging a Multithreaded Application with GDB

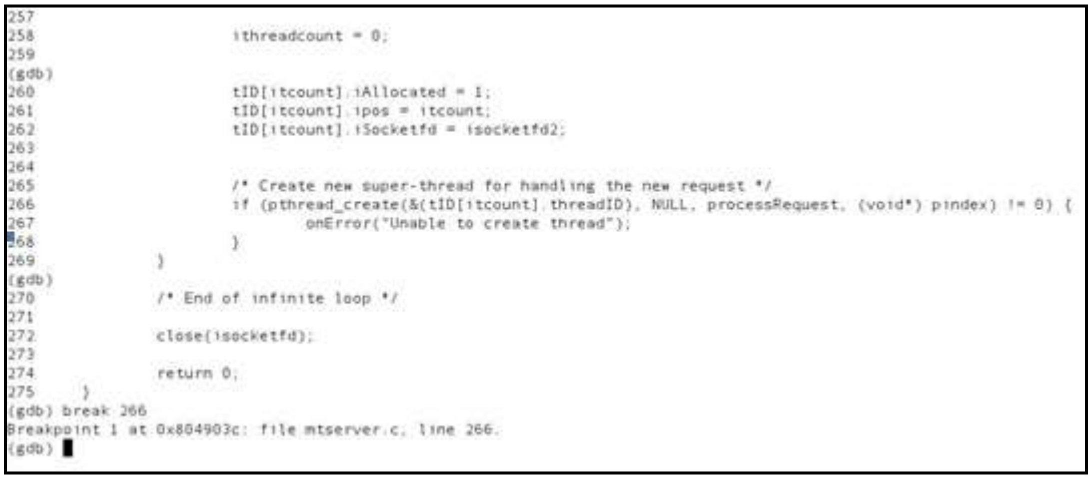
The following steps outline a debug session using GDB:

1.) Start the server within GDB: **`gdb ./server`**.

2.) Set the breakpoints:
   As shown in Figure 3, it makes sense to place the first breakpoint in main(), in front of the function pthread_create(), where the super-threads are created.

   The GDB command **`list main <return>`** lists the source code with line numbers. The first breakpoint is set at line 266 by typing **`break 266 <return>`**.

[1] http://www.gnu.org/software/gdb/
[2] http://www.roguewave.com/products/totalview.aspx



```
257
258                     ithreadcount = 0;
259
(gdb)
260                     tID[itcount].iAllocated = 1;
261                     tID[itcount].ipos = itcount;
262                     tID[itcount].iSocketfd = isocketfd2;
263
264
265                     /* Create new super-thread for handling the new request */
266                     if (pthread_create(&(tID[itcount].threadID), NULL, processRequest, (void*) pindex) != 0) {
267                             onError("Unable to create thread");
268                     }
269             }
(gdb)
270             /* End of infinite loop */
271
272             close(isocketfd);
273
274             return 0;
275     }
(gdb) break 266
Breakpoint 1 at 0x804903c: file mtserver.c, line 266.
(gdb) █
```

Figure 3: Setting a breakpoint in GDB

Additional breakpoints are placed at line 132 in the function ‘processRequest(),’ where the sub-threads will be spawned, and before the calculation starts at line 34 in the function ‘iterate().’ These breakpoints will stop the process when they are encountered by the debugger.

While GDB allows you to create thread-specific breakpoints, it does not prevent you from setting breakpoints on lines that are normally considered invalid, such as in front of comments or on blank lines. These incorrectly placed breakpoints will be recorded by the debugger but will not have the intended effect of stopping the program.

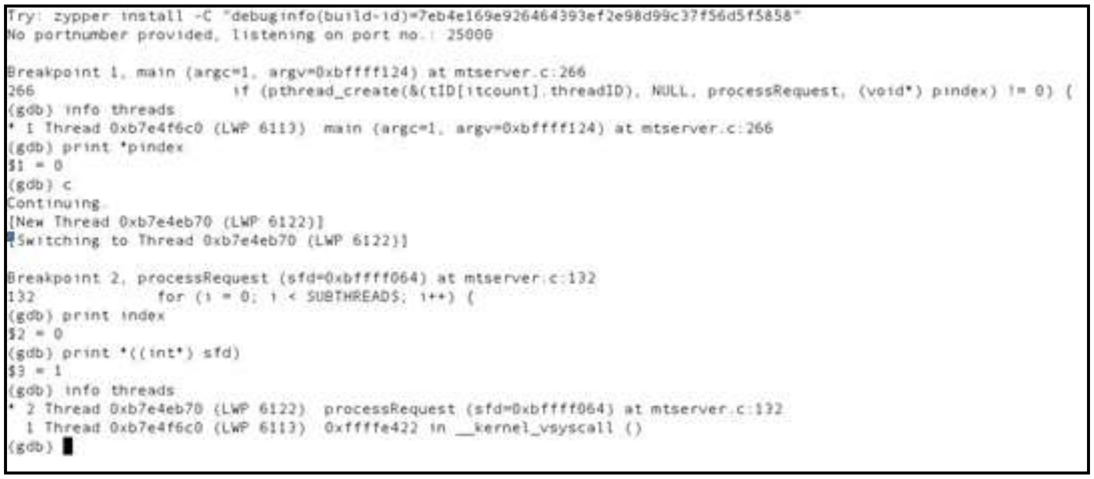
3.) Run the server by typing the GDB command **`run <return>`**.

4.) Start two instances of the client by opening a separate console and typing
`./runclient.sh localhost 25000 1000000 2 <return>`.

The server process runs until it hits breakpoint 1. Type the GDB command **`info threads <return>`** to list the currently running threads. Only the main thread labelled with #1 is listed. Next, check the current value of the variable ***pindex***. This variable points to the index into an array where each thread stores its calculated result and is passed as an argument to the ‘pthread_create()’ function. The GDB command **`print *pindex <return>`** de-references the pointer and displays its contents, currently ‘0.’

5.) Type the GDB command **`c <return>`** to continue execution to the next breakpoint.

6.) As seen in Figure 4, the execution stops at breakpoint 2, line 132, inside of the function processRequest(). The GDB command **`info threads <return>`** shows that two threads are now running, indicating that a second thread, a super-thread, was created in response to a request from the client. It is this super-thread (thread 2) that is stopped at breakpoint 2.



```
Try: zypper install -C "debuginfo(build-id)=7eb4e169e926464393ef2e98d99c37f56d5f5858"
No portnumber provided, listening on port no.: 25000

Breakpoint 1, main (argc=1, argv=0xbffff124) at mtserver.c:266
266                     if (pthread_create(&(tID[itcount].threadID), NULL, processRequest, (void*) pindex) != 0) {
(gdb) info threads
* 1 Thread 0xb7e4f6c0 (LWP 6113)  main (argc=1, argv=0xbffff124) at mtserver.c:266
(gdb) print *pindex
$1 = 0
(gdb) c
Continuing.
[New Thread 0xb7e4eb70 (LWP 6122)]
[Switching to Thread 0xb7e4eb70 (LWP 6122)]

Breakpoint 2, processRequest (sfd=0xbffff064) at mtserver.c:132
132             for (i = 0; i < SUBTHREADS; i++) {
(gdb) print index
$2 = 0
(gdb) print *((int*) sfd)
$3 = 1
(gdb) info threads
* 2 Thread 0xb7e4eb70 (LWP 6122)  processRequest (sfd=0xbffff064) at mtserver.c:132
  1 Thread 0xb7e4f6c0 (LWP 6113)  0xffffe422 in __kernel_vsyscall ()
(gdb) █
```

Figure 4: GDB stopped at the second breakpoint (thread 2 in processRequest())

The GDB **`print index <return>`** command shows the content of the variable ***index*** is set to 0. In contrast, the value the pointer ***sfd*** references has changed to 1. In order to print the value referenced by ***sfd***, it must be cast to an int pointer and de-referenced: (**`print *((int*) sfd) <return>`**).

This appears to be the cause of the incorrect result. The pointer ***sfd*** references the value 1. When this value is passed to the newly-created sub-threads it causes them to write their results to ***sum[1]*** instead of ***sum[0]***.

The value for ***sfd*** was passed to the processRequest() routine from main() via the variable ***pindex***. Looking up the string 'pindex' in main() using the GDB command **`reverse-search pindex <return>`** shows that 'pindex' occurs in lines 252 and 255.

```
251         if (BUG_1) {
252             pindex = &itcount;
253         }
254         else {
255             pindex = &(tID[itcount].ipos);
256         }
```

It is obvious that ***pindex*** accidentally points to the variable ***itcount*** instead of pointing to the already stored index ***ipos*** in the ***tID*** array of structures.

Debugging a Multithreaded Application with TotalView

Next we will take the TotalView Debugger through the same scenario of debugging of a simple, multithreaded application. The TotalView debug session includes the following steps:

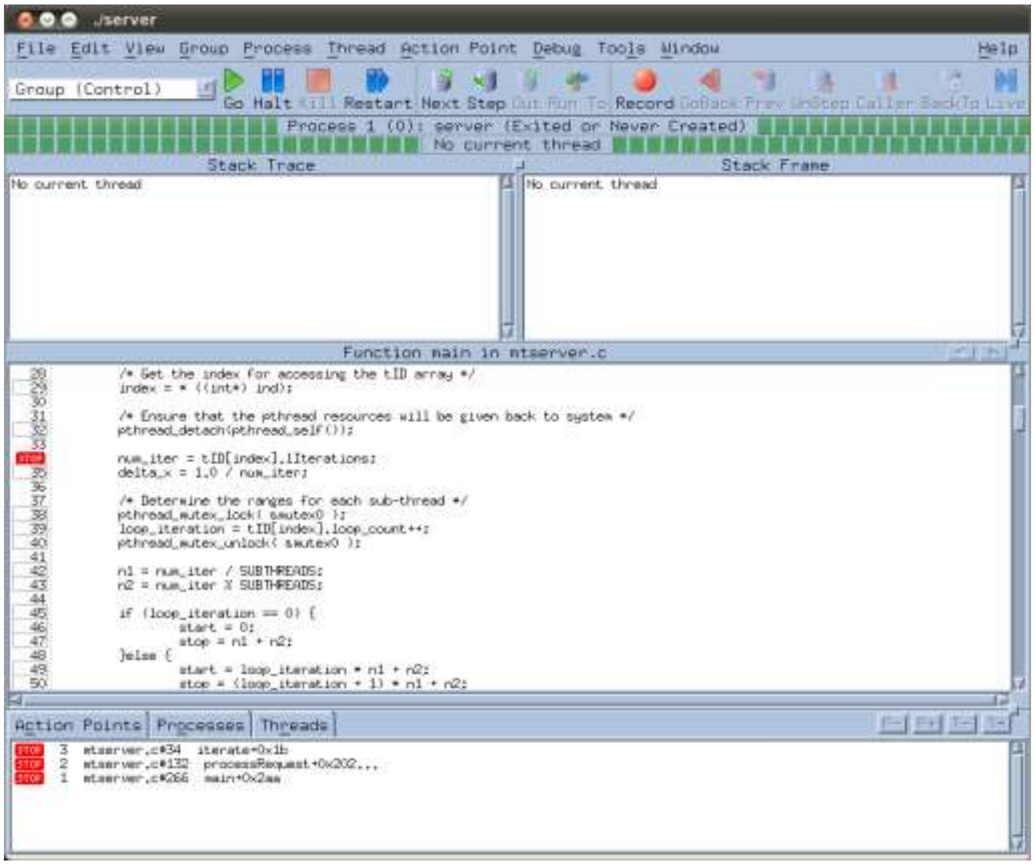
1.) Start the server within TotalView: **`totalview ./server`**.

2.) Set the breakpoints:

Just as was done in the GDB session, the first breakpoint is placed in main(), in front of the function pthread_create(), where the super-threads are created.

Additional breakpoints are placed at line 132 in the function 'processRequest()'and before the calculation starts at line 34 in the function 'iterate().'

Breakpoints in TotalView are a specific type of Action Point. All current Action Points are displayed under the 'Action Points' tab at the bottom of the main window. Clicking on an Action Point in the list displays that line in the source code window. To place a breakpoint in TotalView, simply click on the line number for a valid line in the source code window. Lines that are valid locations for breakpoints are marked with rectangles around the line numbers.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Process 1 (0): server (Exited or Never Created)
No current thread
Stack Trace
No current thread
Stack Frame
No current thread
Function main in mtserver.c
Action Points Processes Threads

```
28          /* Get the index for accessing the tID array */
29          index = * ((int*) ind);
30
31          /* Ensure that the pthread resources will be given back to system */
32          pthread_detach(pthread_self());
33
STOP        num_iter = tID[index].lIterations;
35          delta_x = 1.0 / num_iter;
36
37          /* Determine the ranges for each sub-thread */
38          pthread_mutex_lock( &mutex0 );
39          loop_iteration = tID[index].loop_count++;
40          pthread_mutex_unlock( &mutex0 );
41
42          n1 = num_iter / SUBTHREADS;
43          n2 = num_iter % SUBTHREADS;
44
45          if (loop_iteration == 0) {
46                  start = 0;
47                  stop = n1 + n2;
48          }else {
49                  start = loop_iteration * n1 + n2;
50                  stop = (loop_iteration + 1) * n1 + n2;
```

```
STOP  3  mtserver.c#34   iterate+0x1b
STOP  2  mtserver.c#132  processRequest+0x202...
STOP  1  mtserver.c#266  main+0x2aa
```

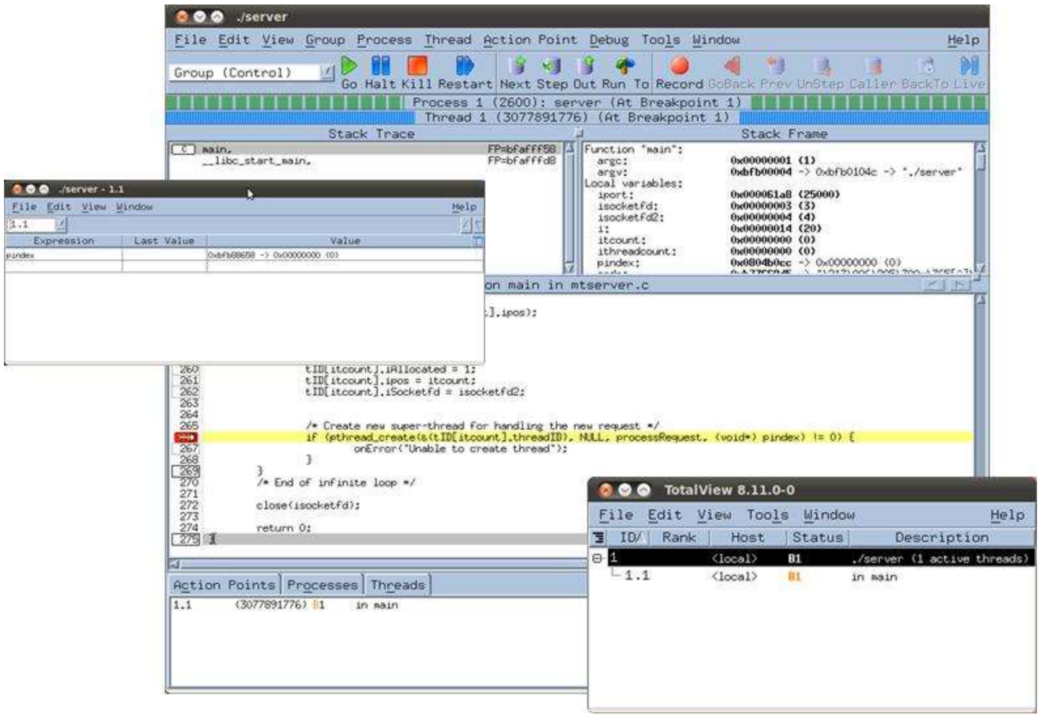
Figure 5: Placing breakpoints in TotalView

3.) **TotalView:** Push the 'Go' button in the toolbar to start the server.

4.) Start two instances of the client by opening a separate console and typing **`./runclient.sh localhost 25000 1000000 2 <return>`**.

When the TotalView process stops at breakpoint 1, the source file containing the breakpoint appears in the source code pane with the breakpoint line highlighted. The main thread is displayed in the Root Window as well as in the tabbed pane. The TotalView thread id is 1.1, the system thread id is displayed in brackets.

To check the value of the variable ***pindex***, move the mouse pointer over the variable name in the source code pane. To continuously monitor the value of any variable, add it to a watch list, called an Expression List, by right-clicking on the variable and selecting 'Add to Expression List' in the pop-up menu. As you can see in Figure 6, items in the Expression List display in a separate, thread-specific window.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Stack Trace
Stack Frame
./server - 1.1
File Edit View Window Help
TotalView 8.11.0-0
File Edit View Tools Window Help
Action Points Processes Threads

```
Process 1 (2600): server (At Breakpoint 1)
Thread 1 (3077891776) (At Breakpoint 1)

main,                 FP=bfafff58
  __libc_start_main,  FP=bfafffd8

Function "main":
  argc:          0x00000001 (1)
  argv:          0xbfb00004 -> 0xbfb0104c -> "./server"
Local variables:
  iport:         0x000061a8 (25000)
  isocketfd:     0x00000003 (3)
  isocketfd2:    0x00000004 (4)
  i:             0x00000014 (20)
  itcount:       0x00000000 (0)
  ithreadcount:  0x00000000 (0)
  pindex:        0x0804b0cc -> 0x00000000 (0)
```

Expression	Last Value	Value
pindex		0xbfb88658 -> 0x00000000 (0)

```
on main in mtserver.c
].ipos);

                tID[itcount].iAllocated = 1;
                tID[itcount].ipos = itcount;
                tID[itcount].iSocketfd = isocketfd2;

                /* Create new super-thread for handling the new request */
                if (pthread_create(&(tID[itcount].threadID), NULL, processRequest, (void*) pindex) != 0) {
                        onError("Unable to create thread");
                }
        }
        /* End of infinite loop */

        close(isocketfd);

        return 0;
}
```

```
1.1        (3077891776) B1      in main
```

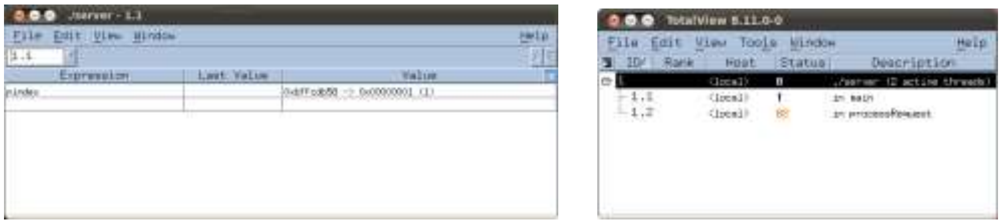
ID	Rank	Host	Status	Description
1		<local>	B1	./server (1 active threads)
1.1		<local>	B1	in main

Figure6: TotalView stopped at the first breakpoint (in main()) with the variable *pindex* displayed in the Expression List

5.) Push the 'Go' button to continue execution to the next breakpoint.

6.) The Root Window shows that the first super-thread is created. The thread is labelled 1.2, and is stopped at breakpoint 2 in the processRequest() function.

As you can see in figure 7, the Expression List shows that ***pindex*** points to a value of 1.



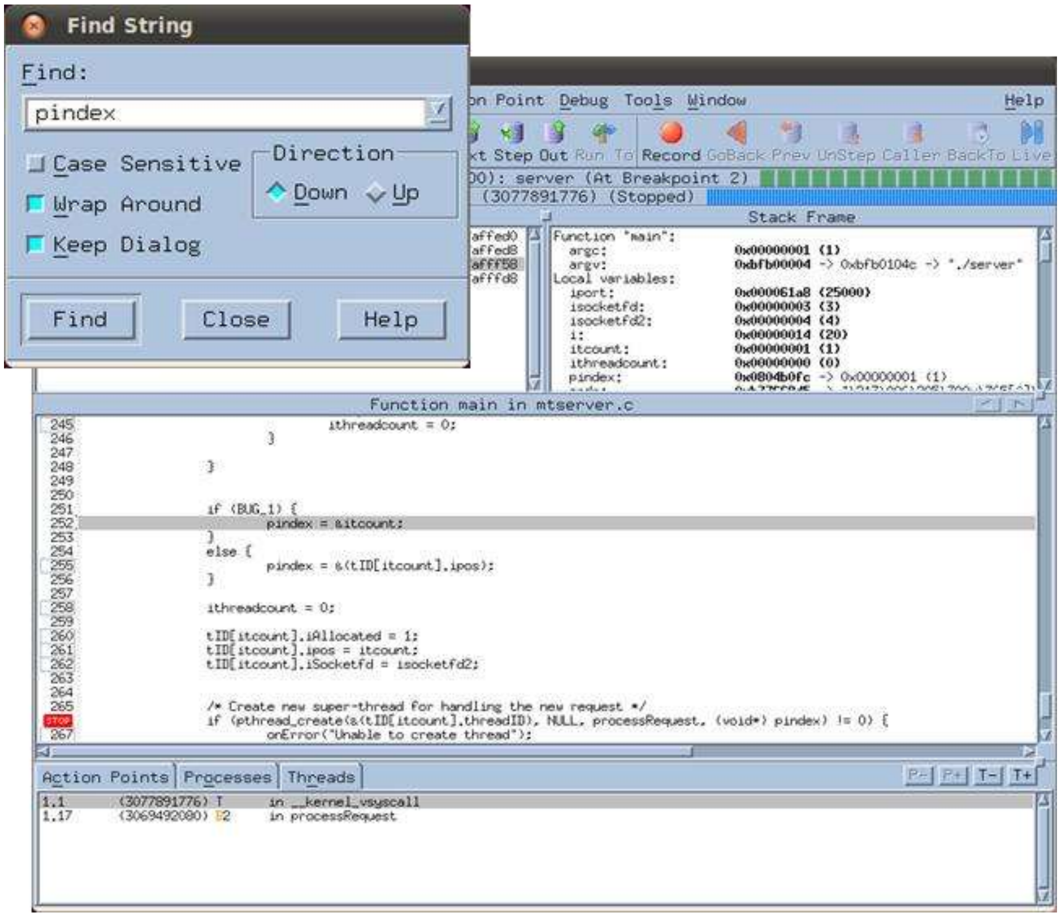
```
./server - 1.1
File  Edit  View  Window                                   Help
1.1
      Expression        Last Value              Value
pindex                                 0xbffcdb58 -> 0x00000001 (1)
```

```
TotalView 8.11.0-0
File  Edit  View  Tools  Window                  Help
 ID   Rank    Host     Status       Description
 1           <local>     B       ./server (2 active threads)
 ├1.1        <local>     T       in main
 └1.2        <local>     B2      in processRequest
```

Figure 7: TotalView Expression List and Root Window after creating the first super-thread

A double-click on thread line 1.1 in the Root Window moves the focus back to the main() function. The source code of main() is displayed in the source code pane.

As you can see in Figure 8,the Edit menu of TotalView's process window contains the option 'Find String' to look up occurrences of the string pindex, which leads to the bug mentioned above.



```
00): server (At Breakpoint 2)
 (3077891776) (Stopped)

affed0
affed8
afff58
afffd8

Function "main":
  argc:                    0x00000001 (1)
  argv:                    0xbfb00004 -> 0xbfb0104c -> "./server"
Local variables:
  iport:                   0x000061a8 (25000)
  isocketfd:               0x00000003 (3)
  isocketfd2:              0x00000004 (4)
  i:                       0x00000014 (20)
  itcount:                 0x00000001 (1)
  ithreadcount:            0x00000000 (0)
  pindex:                  0x0804b0fc -> 0x00000001 (1)
```

```
Function main in mtserver.c
                        ithreadcount = 0;
                }

        }

        if (BUG_1) {
                pindex = &itcount;
        }
        else {
                pindex = &(tID[itcount].ipos);
        }

        ithreadcount = 0;

        tID[itcount].iAllocated = 1;
        tID[itcount].ipos = itcount;
        tID[itcount].iSocketfd = isocketfd2;

        /* Create new super-thread for handling the new request */
STOP    if (pthread_create(&(tID[itcount].threadID), NULL, processRequest, (void*) pindex) != 0) {
                onError("Unable to create thread");
```

```
1.1      (3077891776) T     in __kernel_vsyscall
1.17     (3069492080) B2    in processRequest
```

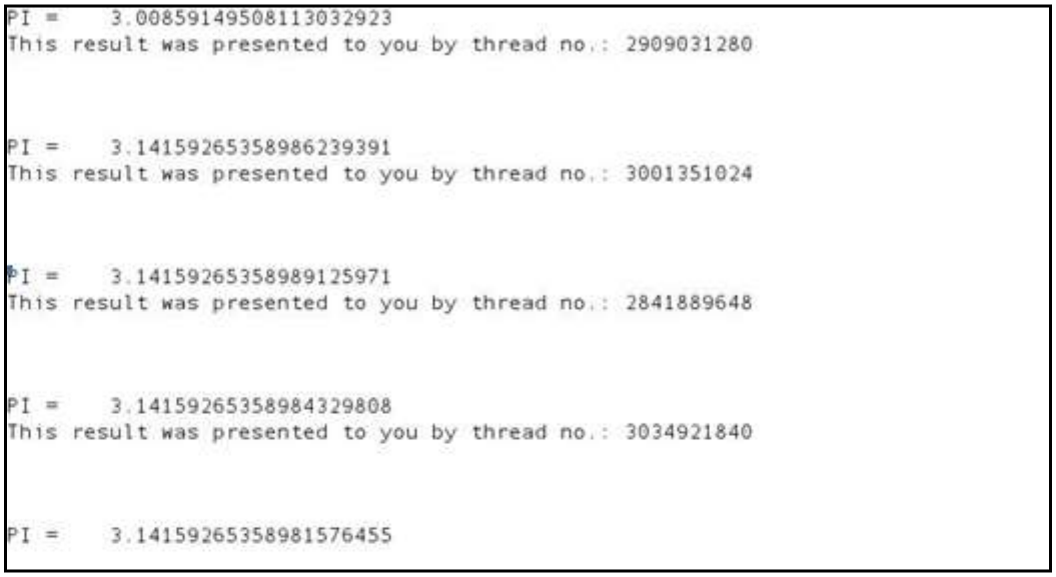
Figure 8: TotalView finding a string in the source code

TotalView's graphical user interface makes it simple to set the breakpoints, look at variable data, and see how the status of the different threads changes.

Investigating a Nondeterministic Issue

We will now move on to investigating another type of debugging challenge that is commonly encountered when developing multithreaded applications - a nondeterministic issue. The server is re-built for the second debugging example with the second implemented bug turned on.

Most requests sent by the client result in correct results, but as you can see in Figure 9, repeated requests produce an intermittent, nondeterministic error in the calculated value for pi.



```
PI =    3.00859149508113032923
This result was presented to you by thread no.: 2909031280



PI =    3.14159265358986239391
This result was presented to you by thread no.: 3001351024



PI =    3.14159265358989125971
This result was presented to you by thread no.: 2841889648



PI =    3.14159265358984329808
This result was presented to you by thread no.: 3034921840



PI =    3.14159265358981576455
```

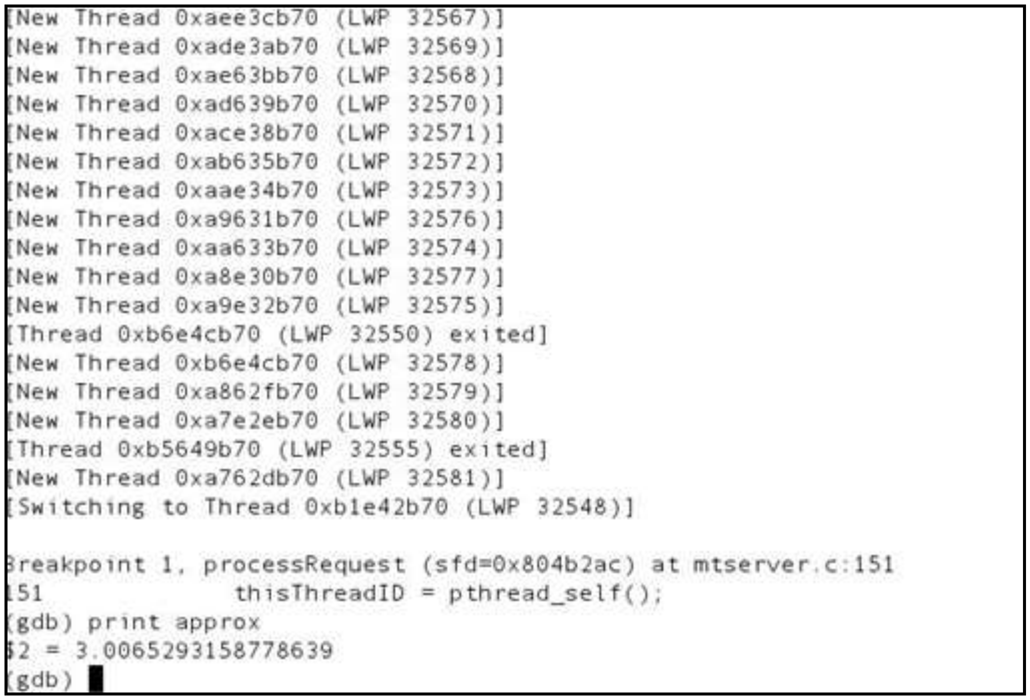
Figure 9: Nondeterministic error

Very often, nondeterministic issues like this are caused by race conditions. An approach in this case is to trigger the issue by placing a conditional breakpoint. The investigation starts by examining how this would be accomplish by using the GDB debugger.

Investigating a Nondeterministic Issue with GDB

1.) After starting the server within GDB, the command **`break 151 if approx < 3.14 <return>`** causes the program to stop at line 151 when the result of ***approx*** is less than 3.14.

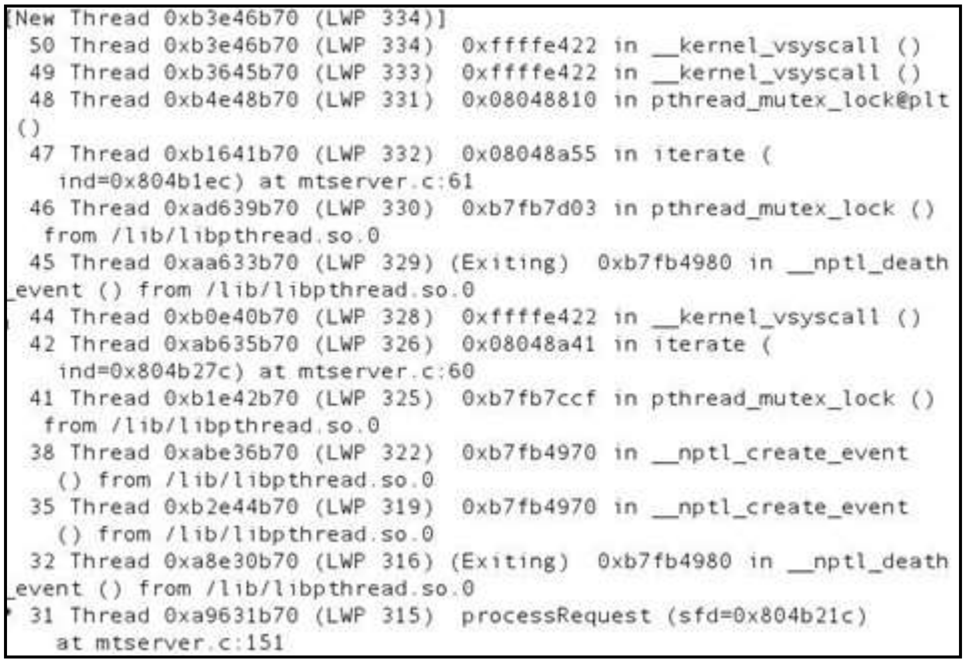
2.) As you can see in Figure 10, running the server several times eventually triggers the breakpoint when the conditional expression is true.



```
[New Thread 0xaee3cb70 (LWP 32567)]
[New Thread 0xade3ab70 (LWP 32569)]
[New Thread 0xae63bb70 (LWP 32568)]
[New Thread 0xad639b70 (LWP 32570)]
[New Thread 0xace38b70 (LWP 32571)]
[New Thread 0xab635b70 (LWP 32572)]
[New Thread 0xaae34b70 (LWP 32573)]
[New Thread 0xa9631b70 (LWP 32576)]
[New Thread 0xaa633b70 (LWP 32574)]
[New Thread 0xa8e30b70 (LWP 32577)]
[New Thread 0xa9e32b70 (LWP 32575)]
[Thread 0xb6e4cb70 (LWP 32550) exited]
[New Thread 0xb6e4cb70 (LWP 32578)]
[New Thread 0xa862fb70 (LWP 32579)]
[New Thread 0xa7e2eb70 (LWP 32580)]
[Thread 0xb5649b70 (LWP 32555) exited]
[New Thread 0xa762db70 (LWP 32581)]
[Switching to Thread 0xb1e42b70 (LWP 32548)]

Breakpoint 1, processRequest (sfd=0x804b2ac) at mtserver.c:151
151             thisThreadID = pthread_self();
(gdb) print approx
$2 = 3.0065293158778639
(gdb) 
```

Figure 10: GDB, breakpoint triggered when a conditional expression is met

3.) As seen in figure 11, to get an idea about what is going on with the referring sub-threads, use the command **`info threads <return>`**.



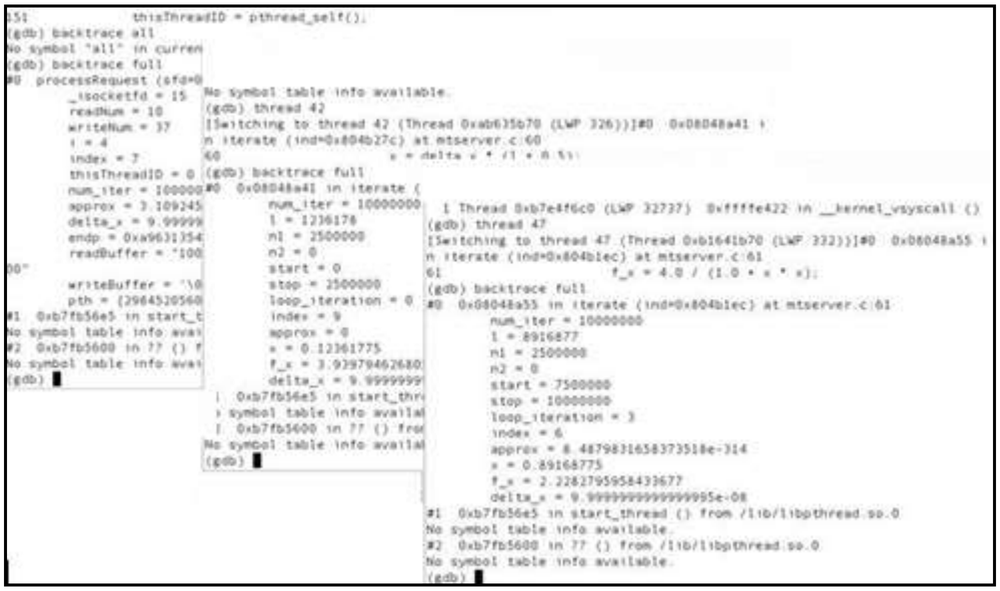
```
[New Thread 0xb3e46b70 (LWP 334)]
  50 Thread 0xb3e46b70 (LWP 334)  0xffffe422 in __kernel_vsyscall ()
  49 Thread 0xb3645b70 (LWP 333)  0xffffe422 in __kernel_vsyscall ()
  48 Thread 0xb4e48b70 (LWP 331)  0x08048810 in pthread_mutex_lock@plt
 ()
  47 Thread 0xb1641b70 (LWP 332)  0x08048a55 in iterate (
    ind=0x804b1ec) at mtserver.c:61
  46 Thread 0xad639b70 (LWP 330)  0xb7fb7d03 in pthread_mutex_lock ()
   from /lib/libpthread.so.0
  45 Thread 0xaa633b70 (LWP 329) (Exiting)  0xb7fb4980 in __nptl_death
_event () from /lib/libpthread.so.0
  44 Thread 0xb0e40b70 (LWP 328)  0xffffe422 in __kernel_vsyscall ()
  42 Thread 0xab635b70 (LWP 326)  0x08048a41 in iterate (
    ind=0x804b27c) at mtserver.c:60
  41 Thread 0xb1e42b70 (LWP 325)  0xb7fb7ccf in pthread_mutex_lock ()
   from /lib/libpthread.so.0
  38 Thread 0xabe36b70 (LWP 322)  0xb7fb4970 in __nptl_create_event
    () from /lib/libpthread.so.0
  35 Thread 0xb2e44b70 (LWP 319)  0xb7fb4970 in __nptl_create_event
    () from /lib/libpthread.so.0
  32 Thread 0xa8e30b70 (LWP 316) (Exiting)  0xb7fb4980 in __nptl_death
_event () from /lib/libpthread.so.0
* 31 Thread 0xa9631b70 (LWP 315)  processRequest (sfd=0x804b21c)
    at mtserver.c:151
```

Figure 11: Output from the GDB info threads command

A reasonable hypothesis for the incorrect result is the assumption that a sub-thread was not able to finish its work until the result was sent back to the requestor. In this case this sub-thread would be stuck in the iterate() function.

The status view on the current threads shows that threads 42 and 47 are still in the iterate() function. Do they belong to the current super-thread which has initiated the stop at the breakpoint?

4.) The command **`backtrace full <return>`** shows all local variables of the thread currently in focus. The focus can be switched to another thread by using the command **`thread <threadnumber> <return>`**.
As you can see in Figure 12, examination of the index variables in the two sub-threads shows they do not belong to the super-thread.



```
151             thisThreadID = pthread_self();
(gdb) backtrace all
No symbol "all" in curren
(gdb) backtrace full
#0  processRequest (sfd=0
        _isocketfd = 15
        readNum = 10
        writeNum = 37
        i = 4
        index = 7
        thisThreadID = 0
        num_iter = 100000
        approx = 3.109245
        delta_x = 9.99999
        endp = 0xa9631354
        readBuffer = "100
00"
        writeBuffer = '\0
        pth = {2984520560
#1  0xb7fb56e5 in start_t
No symbol table info avai
#2  0xb7fb5600 in ?? () f
No symbol table info avai
(gdb) █
```

```
No symbol table info available.
(gdb) thread 42
[Switching to thread 42 (Thread 0xab635b70 (LWP 326))]#0  0x08048a41 i
n iterate (ind=0x804b27c) at mtserver.c:60
60                              x = delta_x * (1 + 0.5);
(gdb) backtrace full
#0  0x08048a41 in iterate (
        num_iter = 10000000
        i = 1236178
        n1 = 2500000
        n2 = 0
        start = 0
        stop = 2500000
        loop_iteration = 0
        index = 9
        approx = 0
        x = 0.12361775
        f_x = 3.93979462680
        delta_x = 9.9999999
 1  0xb7fb56e5 in start_thr
 symbol table info availal
 1  0xb7fb5600 in ?? () fro
No symbol table info availal
(gdb) █
```

```
  1 Thread 0xb7e4f6c0 (LWP 32737)  0xffffe422 in __kernel_vsyscall ()
(gdb) thread 47
[Switching to thread 47 (Thread 0xb1641b70 (LWP 332))]#0  0x08048a55 i
n iterate (ind=0x804b1ec) at mtserver.c:61
61                              f_x = 4.0 / (1.0 + x * x);
(gdb) backtrace full
#0  0x08048a55 in iterate (ind=0x804b1ec) at mtserver.c:61
        num_iter = 10000000
        i = 8916877
        n1 = 2500000
        n2 = 0
        start = 7500000
        stop = 10000000
        loop_iteration = 3
        index = 6
        approx = 8.4879831658373518e-314
        x = 0.89168775
        f_x = 2.2282795958433677
        delta_x = 9.9999999999999995e-08
#1  0xb7fb56e5 in start_thread () from /lib/libpthread.so.0
No symbol table info available.
#2  0xb7fb5600 in ?? () from /lib/libpthread.so.0
No symbol table info available.
(gdb) █
```

Figure 12: Comparing local variables of different threads in GDB

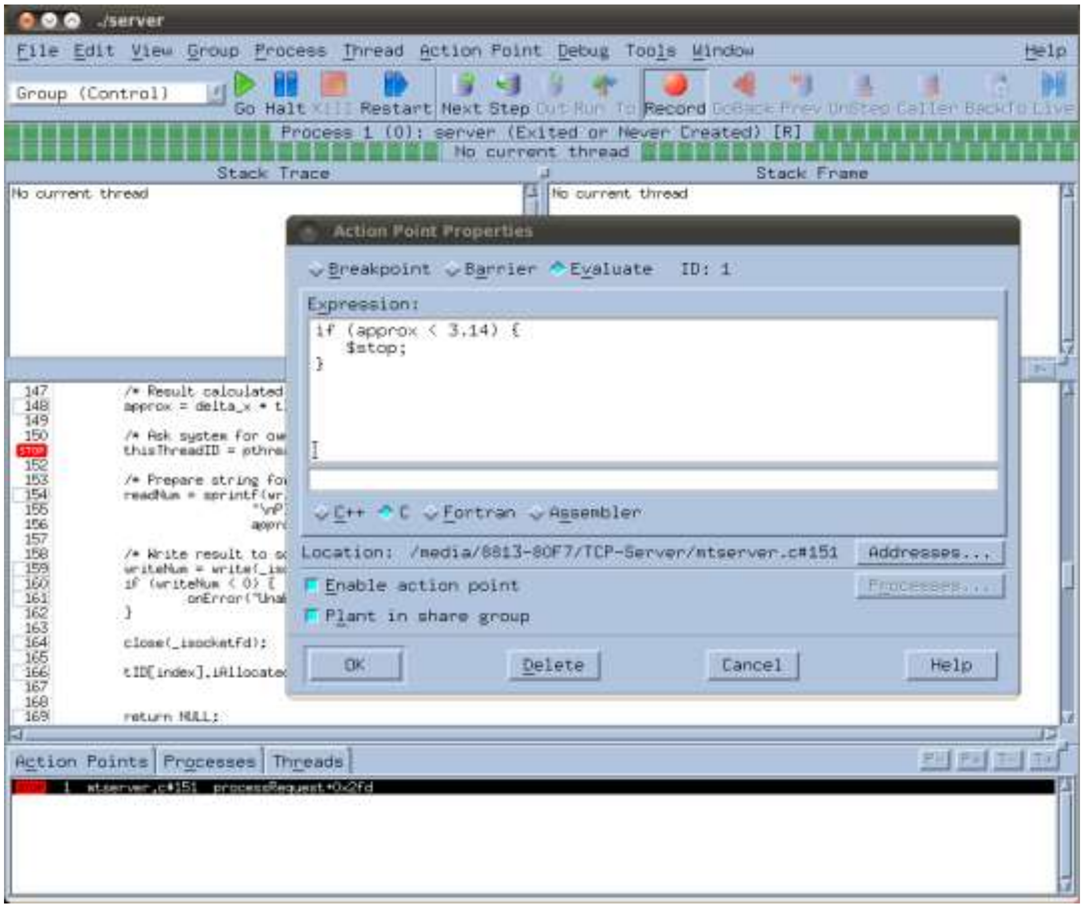
The next possible candidates are threads 41, 46, and 48. These threads are in the pthread_mutex_lock() function that is called inside of iterate(). Switching the focus to these threads using the GDB command **`thread <threadnumber> <return>`**, moving up the backtrace until inside iterate() using **up <return>**, and displaying the local variables using **`backtrace full <return>`** still does not uncover the thread that caused the error.

The only way to solve this issue seems to be by placing breakpoints at the end of function iterate(), running the process until the last thread hits the breakpoint, and trying to determine the referring super-thread. This is very time consuming because of the error's sporadic occurrence and GDB's limited support for solving nondeterministic issues. The issue may not be solved in a reasonable timeframe because the iterate function may be called many times before the error manifests. Subsequent thread activity may also change the critical state, obscuring the clues needed to make a clear diagnosis. Now we will examine how this works with TotalView.

Investigating a Nondeterministic Issue with TotalView

The following steps demonstrate TotalView's investigation of the same issue.

1.) Analogous to the technique used with GDB, a conditional breakpoint is placed at line 151. As you can see in Figure 13, the process stops when ***approx*** `< 3.14`.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Stack Trace
Stack Frame
Action Point Properties
Action Points Processes Threads

```
Process 1 (0): server (Exited or Never Created) [R]
No current thread
```

Stack Trace: No current thread

Stack Frame: No current thread

```
147         /* Result calculated
148         approx = delta_x * t
149
150         /* Ask system for ow
STOP        thisThreadID = pthre
152
153         /* Prepare string fo
154         readNum = sprintf(wr
155                         "\nP
156                         appr
157
158         /* Write result to s
159         writeNum = write(_is
160         if (writeNum < 0) {
161                 onError("Una
162         }
163
164         close(_isocketfd);
165
166         tID[index].iAllocate
167
168
169         return NULL;
```

Action Point Properties

Breakpoint [✓ Evaluate] Barrier   ID: 1

Expression:

```
if (approx < 3.14) {
    $stop;
}
```

C++ [✓ C] Fortran Assembler

Location: /media/8813-80F7/TCP-Server/mtserver.c#151   Addresses...

[✓] Enable action point   Processes...

[✓] Plant in share group

OK   Delete   Cancel   Help

```
1   mtserver.c#151   processRequest+0x2fd
```

Figure 13: TotalView conditional breakpoint configuration

2.) As shown in Figure 14, turn on the TotalView reverse debugging feature called ReplayEngine to record the debugging session. ReplayEngine records the execution history of a program and makes it available for examination. It allows stepping back to any line in the code that was executed while recording was enabled, restoring the state of the entire process to what it was when that line was executed. While very handy in any debugging session, this functionality is indispensable for non-deterministic problems, such as this one.

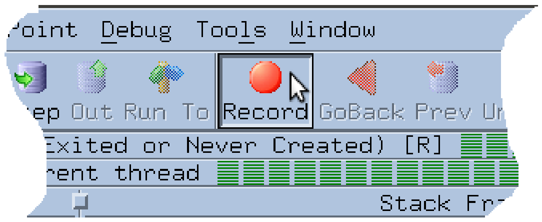
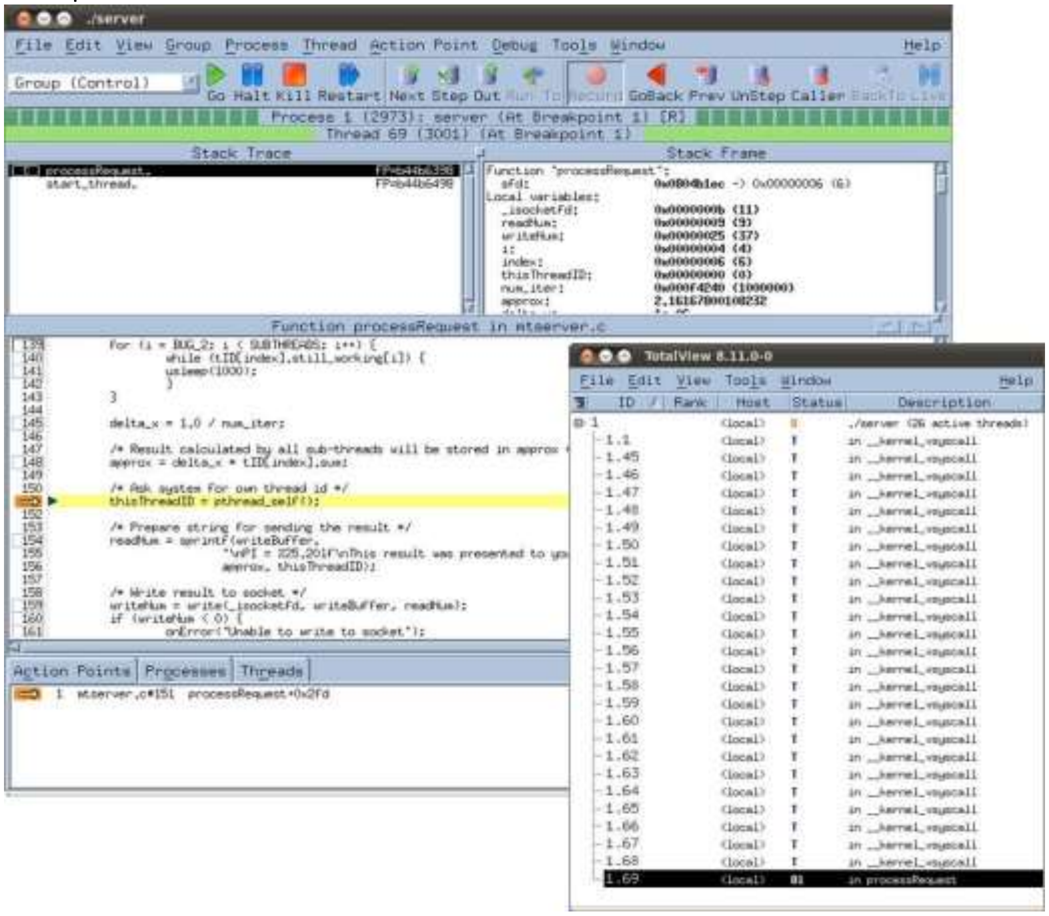

Point Debug Tools Window
ep Out Run To Record GoBack Prev Un
Exited or Never Created) [R]
rent thread
Stack Fr

Figure 14: Switching on the TotalView reverse debugging feature, ReplayEngine

3.) Push the 'Go' button to start the server and send a number of client requests. As you can see in Figure 15, the process stops because a super-thread met the condition at the conditional breakpoint.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Stack Trace
Stack Frame
Action Points Processes Threads
TotalView 8.11.0-0
File Edit View Tools Window Help

```
Process 1 (2973): server (At Breakpoint 1) [R]
Thread 69 (3001) (At Breakpoint 1)
```

```
processRequest,        FP=b44b6398
  start_thread,        FP=b44b6498
```

```
Function "processRequest":
  sFd:                 0x0804b1ec -> 0x00000006 (6)
Local variables:
  _isocketFd:          0x0000000b (11)
  readNum:             0x00000009 (9)
  writeNum:            0x00000025 (37)
  i:                   0x00000004 (4)
  index:               0x00000006 (6)
  thisThreadID:        0x00000000 (0)
  num_iter:            0x000F4240 (1000000)
  approx:              2,16167800108232
```

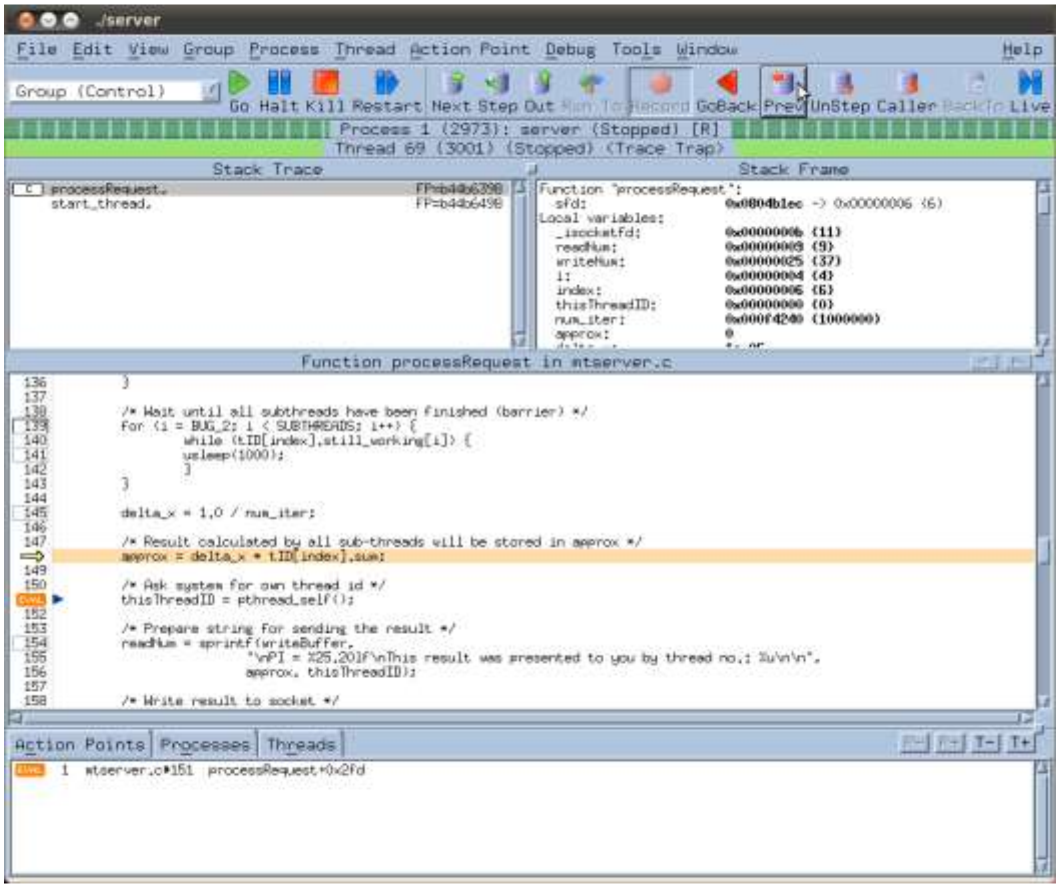
```
Function processRequest in mtserver.c
139         for (i = BUG_2; i < SUBTHREADS; i++) {
140                 while (tID[index].still_working[i]) {
141                 usleep(1000);
142                 }
143         }
144
145         delta_x = 1.0 / num_iter;
146
147         /* Result calculated by all sub-threads will be stored in approx
148         approx = delta_x * tID[index].sum;
149
150         /* Ask system for own thread id */
151  ▶      thisThreadID = pthread_self();
152
153         /* Prepare string for sending the result */
154         readNum = sprintf(writeBuffer,
155                         "\nPI = %25.20lf\nThis result was presented to yo
156                         approx, thisThreadID);
157
158         /* Write result to socket */
159         writeNum = write(_isocketFd, writeBuffer, readNum);
160         if (writeNum < 0) {
161                 onError("Unable to write to socket");
```

```
1  mtserver.c#151  processRequest+0x2fd
```

```
ID        Rank   Host      Status   Description
1                <local>   B        ./server (26 active threads)
  1.1            <local>   T        in __kernel_vsyscall
  1.45           <local>   T        in __kernel_vsyscall
  1.46           <local>   T        in __kernel_vsyscall
  1.47           <local>   T        in __kernel_vsyscall
  1.48           <local>   T        in __kernel_vsyscall
  1.49           <local>   T        in __kernel_vsyscall
  1.50           <local>   T        in __kernel_vsyscall
  1.51           <local>   T        in __kernel_vsyscall
  1.52           <local>   T        in __kernel_vsyscall
  1.53           <local>   T        in __kernel_vsyscall
  1.54           <local>   T        in __kernel_vsyscall
  1.55           <local>   T        in __kernel_vsyscall
  1.56           <local>   T        in __kernel_vsyscall
  1.57           <local>   T        in __kernel_vsyscall
  1.58           <local>   T        in __kernel_vsyscall
  1.59           <local>   T        in __kernel_vsyscall
  1.60           <local>   T        in __kernel_vsyscall
  1.61           <local>   T        in __kernel_vsyscall
  1.62           <local>   T        in __kernel_vsyscall
  1.63           <local>   T        in __kernel_vsyscall
  1.64           <local>   T        in __kernel_vsyscall
  1.65           <local>   T        in __kernel_vsyscall
  1.66           <local>   T        in __kernel_vsyscall
  1.67           <local>   T        in __kernel_vsyscall
  1.68           <local>   T        in __kernel_vsyscall
  1.69           <local>   B1       in processRequest
```

Figure 15: TotalView stops at the conditional breakpoint

4.) As seen in Figure 16, use the TotalView ReplayEngine to step back to line 148 and restore the entire process to its state when that line was executed. Examine the status of each thread at the moment the error occurred.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Stack Trace
Stack Frame
Action Points Processes Threads

```
Process 1 (2973): server (Stopped) [R]
Thread 69 (3001) (Stopped) <Trace Trap>
```

```
processRequest,        FP=b44b6398
start_thread,          FP=b44b6498
```

```
Function "processRequest":
  sfd:                 0x0804b1ec -> 0x00000006 (6)
Local variables:
  _isocketfd:          0x0000000b (11)
  readNum:             0x00000009 (9)
  writeNum:            0x00000025 (37)
  i:                   0x00000004 (4)
  index:               0x00000006 (6)
  thisThreadID:        0x00000000 (0)
  num_iter:            0x000f4240 (1000000)
  approx:              0
```

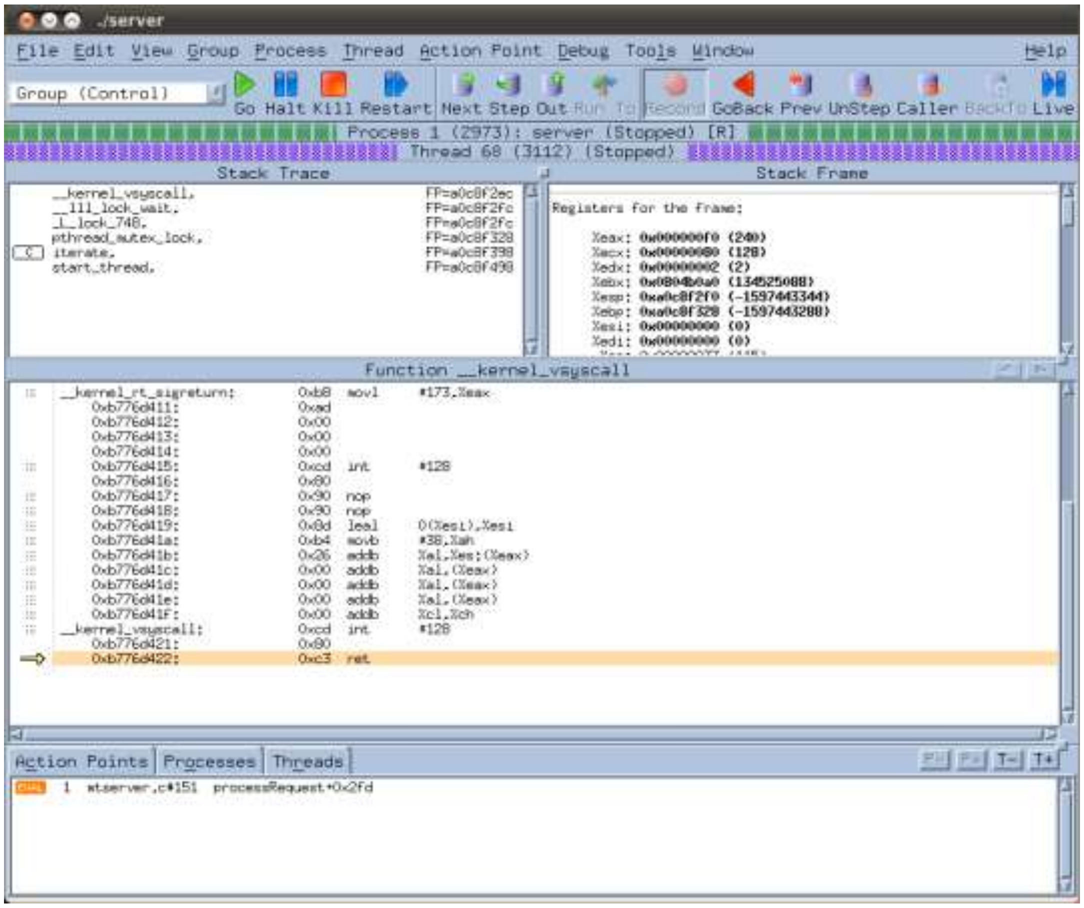
Function processRequest in mtserver.c

```
        }

        /* Wait until all subthreads have been finished (barrier) */
        for (i = BUG_2; i < SUBTHREADS; i++) {
                while (tID[index].still_working[i]) {
                usleep(1000);
                }
        }

        delta_x = 1.0 / num_iter;

        /* Result calculated by all sub-threads will be stored in approx */
        approx = delta_x * tID[index].sum;

        /* Ask system for own thread id */
        thisThreadID = pthread_self();

        /* Prepare string for sending the result */
        readNum = sprintf(writeBuffer,
                     "\nPI = %25.20lf\nThis result was presented to you by thread no.: %u\n\n",
                     approx, thisThreadID);

        /* Write result to socket */
```

```
1  mtserver.c#151  processRequest+0x2fd
```

Figure 16: Stepping backwards using the TotalView ReplayEngine

5.) Following the same hypothesis as in the GDB session, test to see if one of these threads is still executing in the iterate() function. The value of its local variable ***index*** should be 6 because that is the value of the referring super-thread's ***index*** variable. Unfortunately, no thread seems to be currently running in iterate(), but, as you can see in Figure 17, the Stack Trace pane shows that __kernel_vsyscall() was called from within iterate() by several threads.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Stack Trace
Stack Frame
Action Points Processes Threads

```
Process 1 (2973): server (Stopped) [R]
Thread 68 (3112) (Stopped)
```

```
    __kernel_vsyscall,        FP=a0c8f2ec
    __lll_lock_wait,          FP=a0c8f2fc
    _L_lock_748,              FP=a0c8f2fc
    pthread_mutex_lock,       FP=a0c8f328
C   iterate,                  FP=a0c8f398
    start_thread,             FP=a0c8f498
```

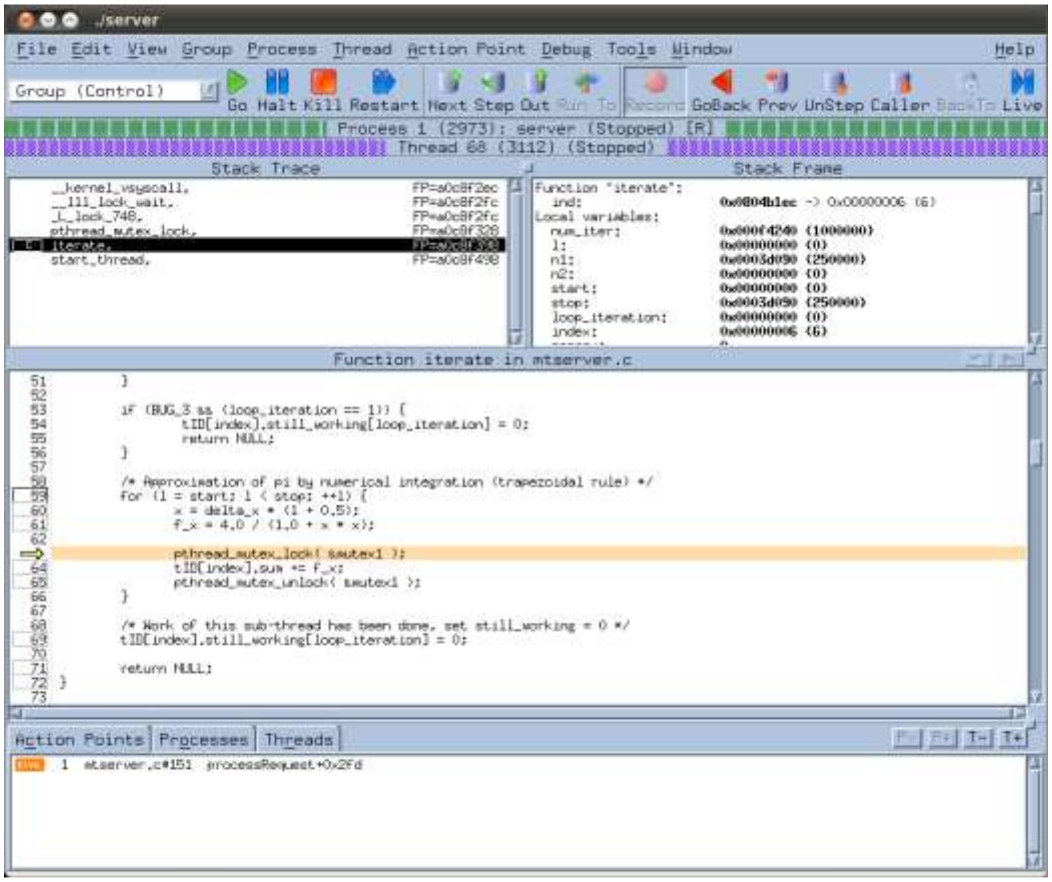
```
Registers for the frame:

    %eax: 0x000000f0 (240)
    %ecx: 0x00000080 (128)
    %edx: 0x00000002 (2)
    %ebx: 0x0804b0a0 (134525088)
    %esp: 0xa0c8f2f0 (-1597443344)
    %ebp: 0xa0c8f328 (-1597443288)
    %esi: 0x00000000 (0)
    %edi: 0x00000000 (0)
```

```
Function __kernel_vsyscall

  __kernel_rt_sigreturn:      0xb8  movl   #173,%eax
      0xb776d411:             0xad
      0xb776d412:             0x00
      0xb776d413:             0x00
      0xb776d414:             0x00
      0xb776d415:             0xcd  int    #128
      0xb776d416:             0x80
      0xb776d417:             0x90  nop
      0xb776d418:             0x90  nop
      0xb776d419:             0x8d  leal   0(%esi),%esi
      0xb776d41a:             0xb4  movb   #38,%ah
      0xb776d41b:             0x26  addb   %al,%es:(%eax)
      0xb776d41c:             0x00  addb   %al,(%eax)
      0xb776d41d:             0x00  addb   %al,(%eax)
      0xb776d41e:             0x00  addb   %al,(%eax)
      0xb776d41f:             0x00  addb   %cl,%ch
  __kernel_vsyscall:          0xcd  int    #128
      0xb776d421:             0x80
=>    0xb776d422:             0xc3  ret
```

```
EVAL  1  mtserver.c#151  processRequest+0x2fd
```

Figure 17: Thread in __kernel_vsyscall() in TotalView

6.) Click on ‘iterate’ in the Stack Trace pane and check the local variables in the referring Stack Frame pane to see that thread 68, which belongs to the super-thread at the breakpoint, has not yet finished the iterate() function. As you can see in Figure 18, this is the reason for the incorrect result.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Action Points Processes Threads

```
Process 1 (2973): server (Stopped) [R]
Thread 68 (3112) (Stopped)
```

```
Stack Trace
    __kernel_vsyscall,        FP=a0c8f2ec
    __lll_lock_wait,          FP=a0c8f2fc
    _L_lock_748,              FP=a0c8f2fc
    pthread_mutex_lock,       FP=a0c8f328
 C  iterate,                  FP=a0c8f398
    start_thread,             FP=a0c8f498
```

```
Stack Frame
Function "iterate":
  ind:                0x0804b1ec -> 0x00000006 (6)
Local variables:
  num_iter:           0x000f4240 (1000000)
  l:                  0x00000000 (0)
  n1:                 0x0003d090 (250000)
  n2:                 0x00000000 (0)
  start:              0x00000000 (0)
  stop:               0x0003d090 (250000)
  loop_iteration:     0x00000000 (0)
  index:              0x00000006 (6)
```

```
Function iterate in mtserver.c
        }

        if (BUG_3 && (loop_iteration == 1)) {
                tID[index].still_working[loop_iteration] = 0;
                return NULL;
        }

        /* Approximation of pi by numerical integration (trapezoidal rule) */
        for (l = start; l < stop; ++l) {
                x = delta_x * (l + 0.5);
                f_x = 4.0 / (1.0 + x * x);

                pthread_mutex_lock( &mutex1 );
                tID[index].sum += f_x;
                pthread_mutex_unlock( &mutex1 );
        }

        /* Work of this sub-thread has been done, set still_working = 0 */
        tID[index].still_working[loop_iteration] = 0;

        return NULL;
}
```

```
Live  1  mtserver.c#151  processRequest+0x2Fd
```

Figure 18: One sub-thread still executing iterate() in TotalView

The Stack Frame pane for thread 68 shows that the value of variable ***loop_iteration*** is 0. This means the status of this thread is stored at ***tID[6].still_working[0]***.
Using ReplayEngine and stepping the super-thread backwards through the barrier at lines 139 to 143 demonstrates the status of this thread will be disregarded.
The issue has been solved.

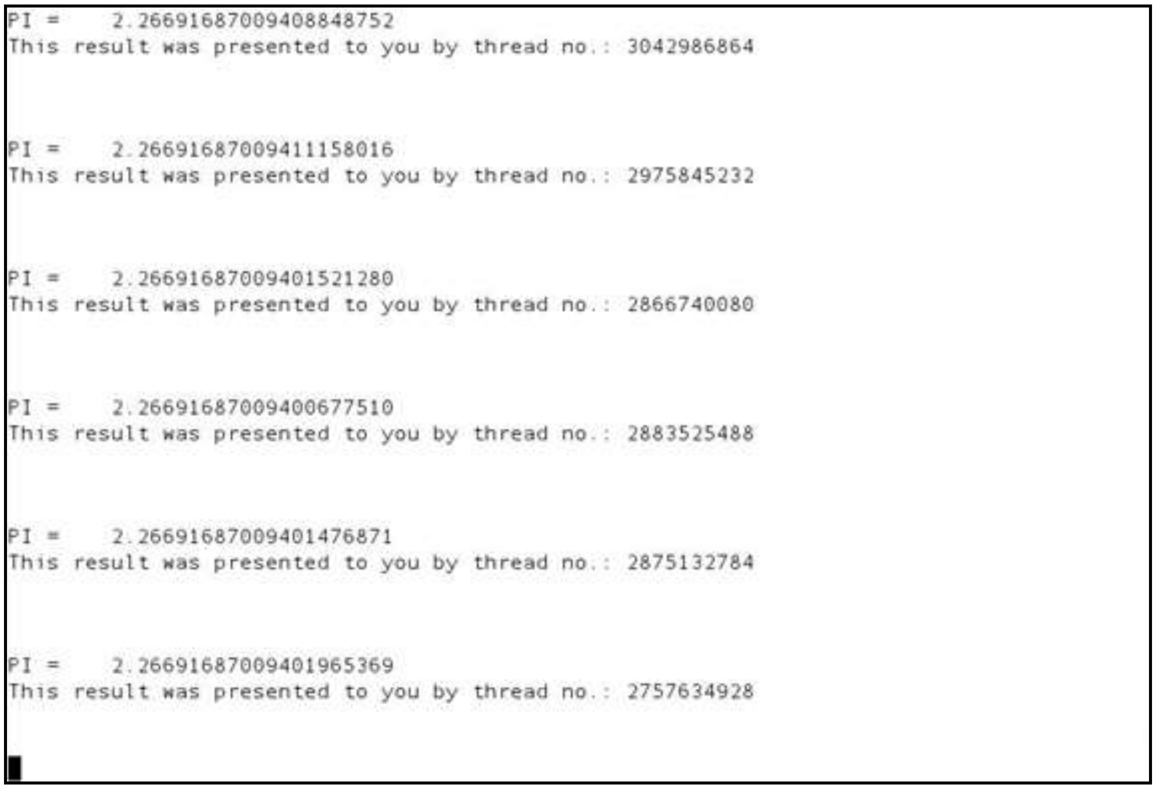
```
    for (i = BUG_2   ; i < SUBTHREADS; i++) {
      while (tID[index].still_working[i]) {
      usleep(1000);
      }
 }
```

The value of i starts at 1 and increments upwards, so the super thread never checks to see if the thread for loop iteration 0 is done.

Notice how straightforwardly we could approach the solution to this issue; having the recorded execution history makes it really easy to check what the program is doing as we work backwards through the logic of the program from the incorrect result to the root cause of the bug.

Thread Control Capabilities of GDB vs. TotalView

For the third debugging session, we rebuild the server with bug 3 enabled.
In this case, as can be seen in Figure 19, the results are identical for every request but the values are always lower than 3.14. The reason for this behaviour could be that one or more sub-threads do not finish the approximation properly.



```
PI =     2.26691687009408848752
This result was presented to you by thread no.: 3042986864


PI =     2.26691687009411158016
This result was presented to you by thread no.: 2975845232


PI =     2.26691687009401521280
This result was presented to you by thread no.: 2866740080


PI =     2.26691687009400677510
This result was presented to you by thread no.: 2883525488


PI =     2.26691687009401476871
This result was presented to you by thread no.: 2875132784


PI =     2.26691687009401965369
This result was presented to you by thread no.: 2757634928
```

Figure 19: Results of a series of requests with BUG_3 enabled

In order to investigate this bug the following approach is planned:

1. Set a breakpoint at the very beginning of the iterate() function such that all sub-threads associated with a particular request are stopped.

2. Perform a few single-steps of these threads, in lockstep, until the calculation of the different iteration() sections start.

3. Move each individual thread in single-step mode in order to analyze its behaviour.

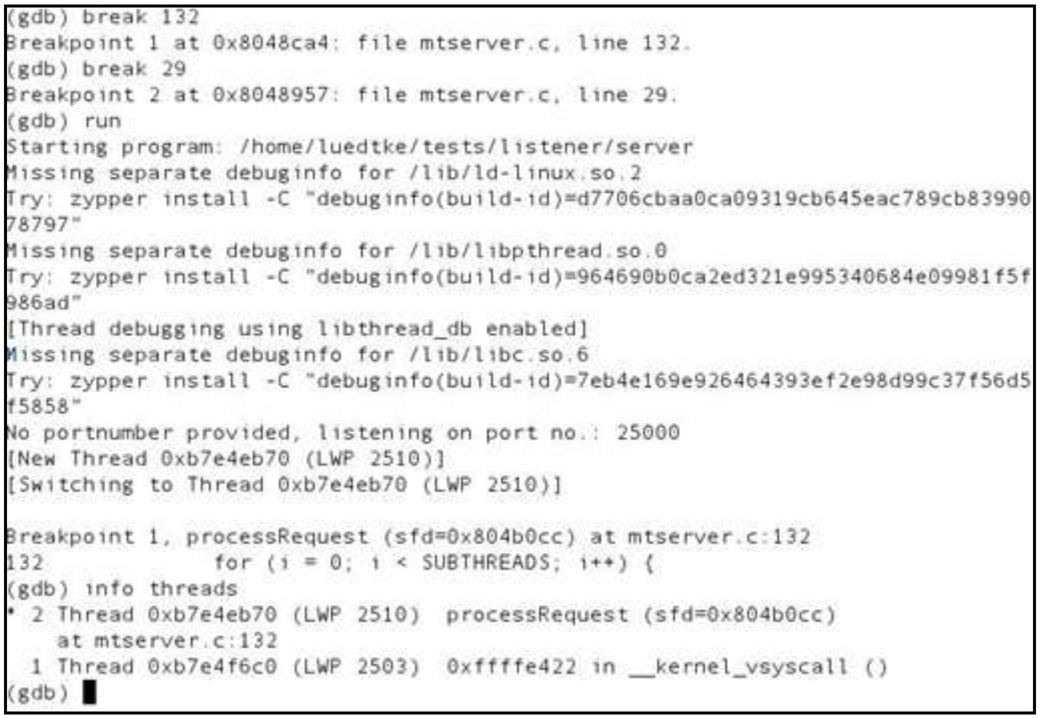
Thread Control Capabilities of GDB

Again, the debug session will first be demonstrated using GDB.

The capability of controlling threads independently is limited in GDB. The ability to run a single thread is dependent on the referring operating system's ability to lock the scheduler. Furthermore, GDB generally stops the entire process when a single thread hits a breakpoint.

1. To deal with these limitations of GDB, the debugging process will start by setting two breakpoints. The first one is placed at line 132 where the sub-threads are created in function processRequest(). The second breakpoint is placed on line 29 at the beginning of the function iterate().The GDB commands **`break 132 <return>`** and **`break 29 <return>`** accomplish this.

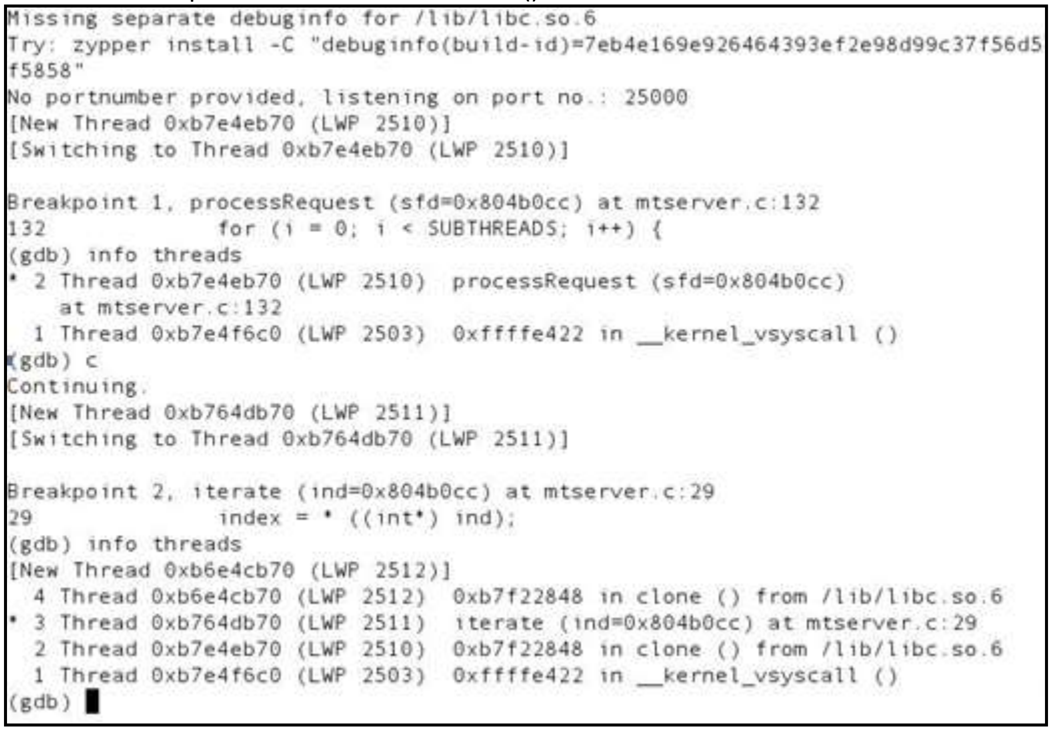
2. Running the client by typing **`./client localhost 25000 1000000 <return>`** or **`./runclient.sh localhost 25000 1000000 1 <return>`** in a separate console window will cause the server to run to the first breakpoint, as seen in Figure 20. The GDB command **`info threads <return>`** helps identify the super-thread.



```
(gdb) break 132
Breakpoint 1 at 0x8048ca4: file mtserver.c, line 132.
(gdb) break 29
Breakpoint 2 at 0x8048957: file mtserver.c, line 29.
(gdb) run
Starting program: /home/luedtke/tests/listener/server
Missing separate debuginfo for /lib/ld-linux.so.2
Try: zypper install -C "debuginfo(build-id)=d7706cbaa0ca09319cb645eac789cb83990
78797"
Missing separate debuginfo for /lib/libpthread.so.0
Try: zypper install -C "debuginfo(build-id)=964690b0ca2ed321e995340684e09981f5f
986ad"
[Thread debugging using libthread_db enabled]
Missing separate debuginfo for /lib/libc.so.6
Try: zypper install -C "debuginfo(build-id)=7eb4e169e926464393ef2e98d99c37f56d5
f5858"
No portnumber provided, listening on port no.: 25000
[New Thread 0xb7e4eb70 (LWP 2510)]
[Switching to Thread 0xb7e4eb70 (LWP 2510)]

Breakpoint 1, processRequest (sfd=0x804b0cc) at mtserver.c:132
132             for (i = 0; i < SUBTHREADS; i++) {
(gdb) info threads
* 2 Thread 0xb7e4eb70 (LWP 2510)  processRequest (sfd=0x804b0cc)
    at mtserver.c:132
  1 Thread 0xb7e4f6c0 (LWP 2503)  0xffffe422 in __kernel_vsyscall ()
(gdb) █
```

Figure 20: GDB stopped at the first breakpoint

3. As seen in Figure 21, typing **continue** **`<return>`**, or **`c <return>`**, will stop the first sub-thread at breakpoint 29 in function "iterate()."



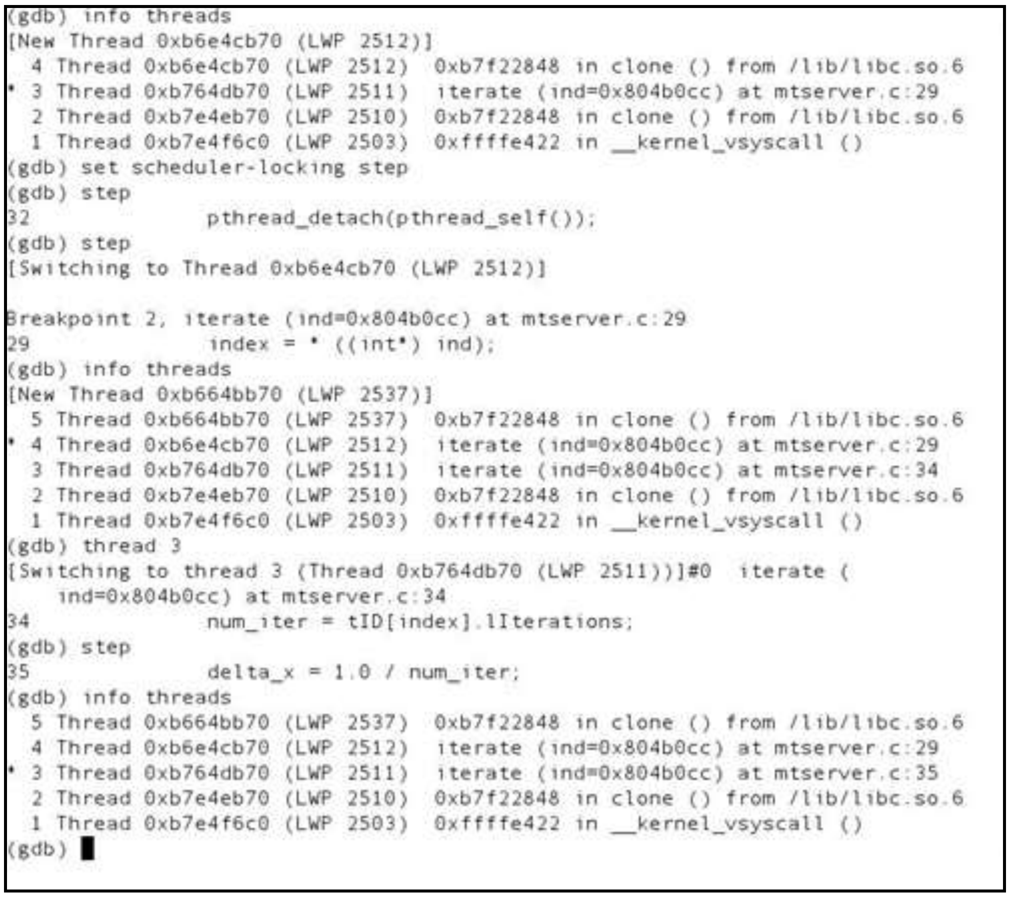
```
Missing separate debuginfo for /lib/libc.so.6
Try: zypper install -C "debuginfo(build-id)=7eb4e169e926464393ef2e98d99c37f56d5
f5858"
No portnumber provided, listening on port no.: 25000
[New Thread 0xb7e4eb70 (LWP 2510)]
[Switching to Thread 0xb7e4eb70 (LWP 2510)]

Breakpoint 1, processRequest (sfd=0x804b0cc) at mtserver.c:132
132             for (i = 0; i < SUBTHREADS; i++) {
(gdb) info threads
* 2 Thread 0xb7e4eb70 (LWP 2510)  processRequest (sfd=0x804b0cc)
    at mtserver.c:132
  1 Thread 0xb7e4f6c0 (LWP 2503)  0xffffe422 in __kernel_vsyscall ()
(gdb) c
Continuing.
[New Thread 0xb764db70 (LWP 2511)]
[Switching to Thread 0xb764db70 (LWP 2511)]

Breakpoint 2, iterate (ind=0x804b0cc) at mtserver.c:29
29              index = * ((int*) ind);
(gdb) info threads
[New Thread 0xb6e4cb70 (LWP 2512)]
  4 Thread 0xb6e4cb70 (LWP 2512)  0xb7f22848 in clone () from /lib/libc.so.6
* 3 Thread 0xb764db70 (LWP 2511)  iterate (ind=0x804b0cc) at mtserver.c:29
  2 Thread 0xb7e4eb70 (LWP 2510)  0xb7f22848 in clone () from /lib/libc.so.6
  1 Thread 0xb7e4f6c0 (LWP 2503)  0xffffe422 in __kernel_vsyscall ()
(gdb) █
```

Figure 21: GDB stopped at the second breakpoint

4. In order to perform single-steps of only thread 3, set the scheduler locking mode to 'step': **`set scheduler-locking step <return>`**.

5. Single-step thread 3 using either the command **`step <return>`** (stepping into functions) or **`next <return>`** (stepping over functions). After executing two single-steps, thread 4 will also enter the function iterate(), but it will stay at breakpoint 2, as seen in Figure 22.

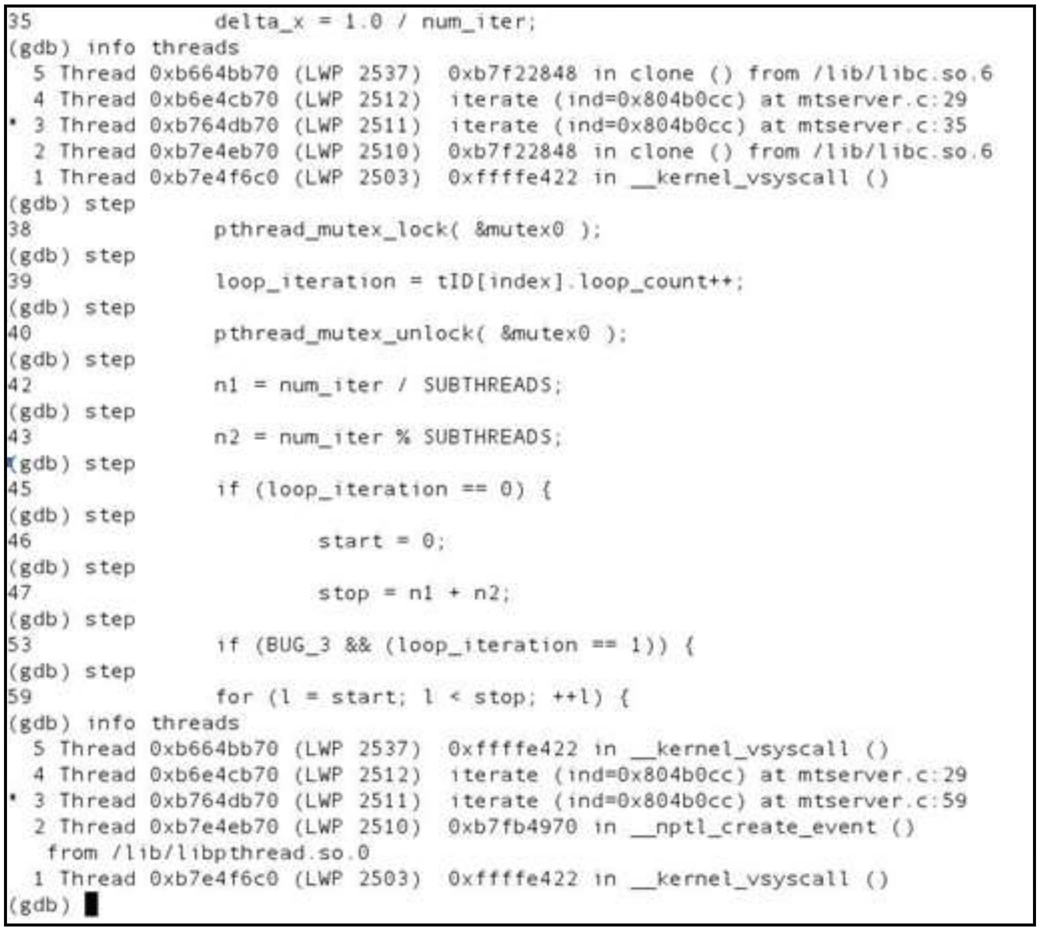
6. The focus has automatically been switched to thread 4, so it has to be manually switched back to thread 3 using the command **`thread 3 <return>`**.



```
(gdb) info threads
[New Thread 0xb6e4cb70 (LWP 2512)]
  4 Thread 0xb6e4cb70 (LWP 2512)  0xb7f22848 in clone () from /lib/libc.so.6
* 3 Thread 0xb764db70 (LWP 2511)  iterate (ind=0x804b0cc) at mtserver.c:29
  2 Thread 0xb7e4eb70 (LWP 2510)  0xb7f22848 in clone () from /lib/libc.so.6
  1 Thread 0xb7e4f6c0 (LWP 2503)  0xffffe422 in __kernel_vsyscall ()
(gdb) set scheduler-locking step
(gdb) step
32              pthread_detach(pthread_self());
(gdb) step
[Switching to Thread 0xb6e4cb70 (LWP 2512)]

Breakpoint 2, iterate (ind=0x804b0cc) at mtserver.c:29
29              index = * ((int*) ind);
(gdb) info threads
[New Thread 0xb664bb70 (LWP 2537)]
  5 Thread 0xb664bb70 (LWP 2537)  0xb7f22848 in clone () from /lib/libc.so.6
* 4 Thread 0xb6e4cb70 (LWP 2512)  iterate (ind=0x804b0cc) at mtserver.c:29
  3 Thread 0xb764db70 (LWP 2511)  iterate (ind=0x804b0cc) at mtserver.c:34
  2 Thread 0xb7e4eb70 (LWP 2510)  0xb7f22848 in clone () from /lib/libc.so.6
  1 Thread 0xb7e4f6c0 (LWP 2503)  0xffffe422 in __kernel_vsyscall ()
(gdb) thread 3
[Switching to thread 3 (Thread 0xb764db70 (LWP 2511))]#0  iterate (
    ind=0x804b0cc) at mtserver.c:34
34              num_iter = tID[index].lIterations;
(gdb) step
35              delta_x = 1.0 / num_iter;
(gdb) info threads
  5 Thread 0xb664bb70 (LWP 2537)  0xb7f22848 in clone () from /lib/libc.so.6
  4 Thread 0xb6e4cb70 (LWP 2512)  iterate (ind=0x804b0cc) at mtserver.c:29
* 3 Thread 0xb764db70 (LWP 2511)  iterate (ind=0x804b0cc) at mtserver.c:35
  2 Thread 0xb7e4eb70 (LWP 2510)  0xb7f22848 in clone () from /lib/libc.so.6
  1 Thread 0xb7e4f6c0 (LWP 2503)  0xffffe422 in __kernel_vsyscall ()
(gdb) █
```

Figure 22: GDB after performing two single-steps

7. Single-stepping thread 3 until line 59, where the approximation starts, does not show any unusual behaviour. Therefore, switch focus to thread 4 in order to single-step this thread up to line 59, as seen in Figure 23.



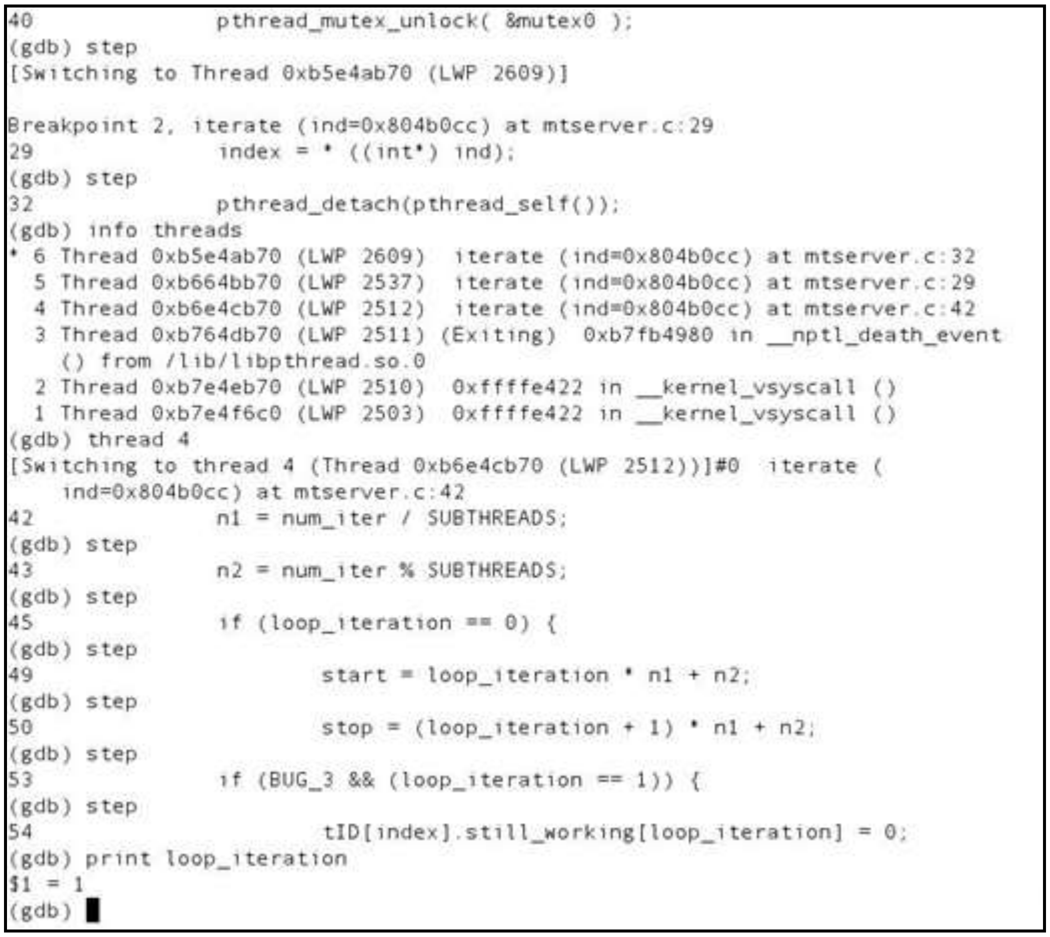
```
35              delta_x = 1.0 / num_iter;
(gdb) info threads
  5 Thread 0xb664bb70 (LWP 2537)  0xb7f22848 in clone () from /lib/libc.so.6
  4 Thread 0xb6e4cb70 (LWP 2512)  iterate (ind=0x804b0cc) at mtserver.c:29
* 3 Thread 0xb764db70 (LWP 2511)  iterate (ind=0x804b0cc) at mtserver.c:35
  2 Thread 0xb7e4eb70 (LWP 2510)  0xb7f22848 in clone () from /lib/libc.so.6
  1 Thread 0xb7e4f6c0 (LWP 2503)  0xffffe422 in __kernel_vsyscall ()
(gdb) step
38              pthread_mutex_lock( &mutex0 );
(gdb) step
39              loop_iteration = tID[index].loop_count++;
(gdb) step
40              pthread_mutex_unlock( &mutex0 );
(gdb) step
42              n1 = num_iter / SUBTHREADS;
(gdb) step
43              n2 = num_iter % SUBTHREADS;
(gdb) step
45              if (loop_iteration == 0) {
(gdb) step
46                      start = 0;
(gdb) step
47                      stop = n1 + n2;
(gdb) step
53              if (BUG_3 && (loop_iteration == 1)) {
(gdb) step
59              for (l = start; l < stop; ++l) {
(gdb) info threads
  5 Thread 0xb664bb70 (LWP 2537)  0xffffe422 in __kernel_vsyscall ()
  4 Thread 0xb6e4cb70 (LWP 2512)  iterate (ind=0x804b0cc) at mtserver.c:29
* 3 Thread 0xb764db70 (LWP 2511)  iterate (ind=0x804b0cc) at mtserver.c:59
  2 Thread 0xb7e4eb70 (LWP 2510)  0xb7fb4970 in __nptl_create_event ()
   from /lib/libpthread.so.0
  1 Thread 0xb7e4f6c0 (LWP 2503)  0xffffe422 in __kernel_vsyscall ()
(gdb) 
```

Figure 23: Single-stepping thread 3 to line 59 in GDB

8. As shown in Figure 24, single-step thread 4, while taking care that it remains in focus shows that, in contrast to thread 3, it will hit lines 54 and 55, which will cause this thread to terminate.

The bug has been uncovered:

```
if (BUG_3 && (loop_iteration == 1)) {
        tID[index].still_working[loop_iteration] = 0;
         return NULL;
}
```



```
40              pthread_mutex_unlock( &mutex0 );
(gdb) step
[Switching to Thread 0xb5e4ab70 (LWP 2609)]

Breakpoint 2, iterate (ind=0x804b0cc) at mtserver.c:29
29              index = * ((int*) ind);
(gdb) step
32              pthread_detach(pthread_self());
(gdb) info threads
* 6 Thread 0xb5e4ab70 (LWP 2609)  iterate (ind=0x804b0cc) at mtserver.c:32
  5 Thread 0xb664bb70 (LWP 2537)  iterate (ind=0x804b0cc) at mtserver.c:29
  4 Thread 0xb6e4cb70 (LWP 2512)  iterate (ind=0x804b0cc) at mtserver.c:42
  3 Thread 0xb764db70 (LWP 2511) (Exiting)  0xb7fb4980 in __nptl_death_event
    () from /lib/libpthread.so.0
  2 Thread 0xb7e4eb70 (LWP 2510)  0xffffe422 in __kernel_vsyscall ()
  1 Thread 0xb7e4f6c0 (LWP 2503)  0xffffe422 in __kernel_vsyscall ()
(gdb) thread 4
[Switching to thread 4 (Thread 0xb6e4cb70 (LWP 2512))]#0  iterate (
    ind=0x804b0cc) at mtserver.c:42
42              n1 = num_iter / SUBTHREADS;
(gdb) step
43              n2 = num_iter % SUBTHREADS;
(gdb) step
45              if (loop_iteration == 0) {
(gdb) step
49                      start = loop_iteration * n1 + n2;
(gdb) step
50                      stop = (loop_iteration + 1) * n1 + n2;
(gdb) step
53              if (BUG_3 && (loop_iteration == 1)) {
(gdb) step
54                      tID[index].still_working[loop_iteration] = 0;
(gdb) print loop_iteration
$1 = 1
(gdb) █
```

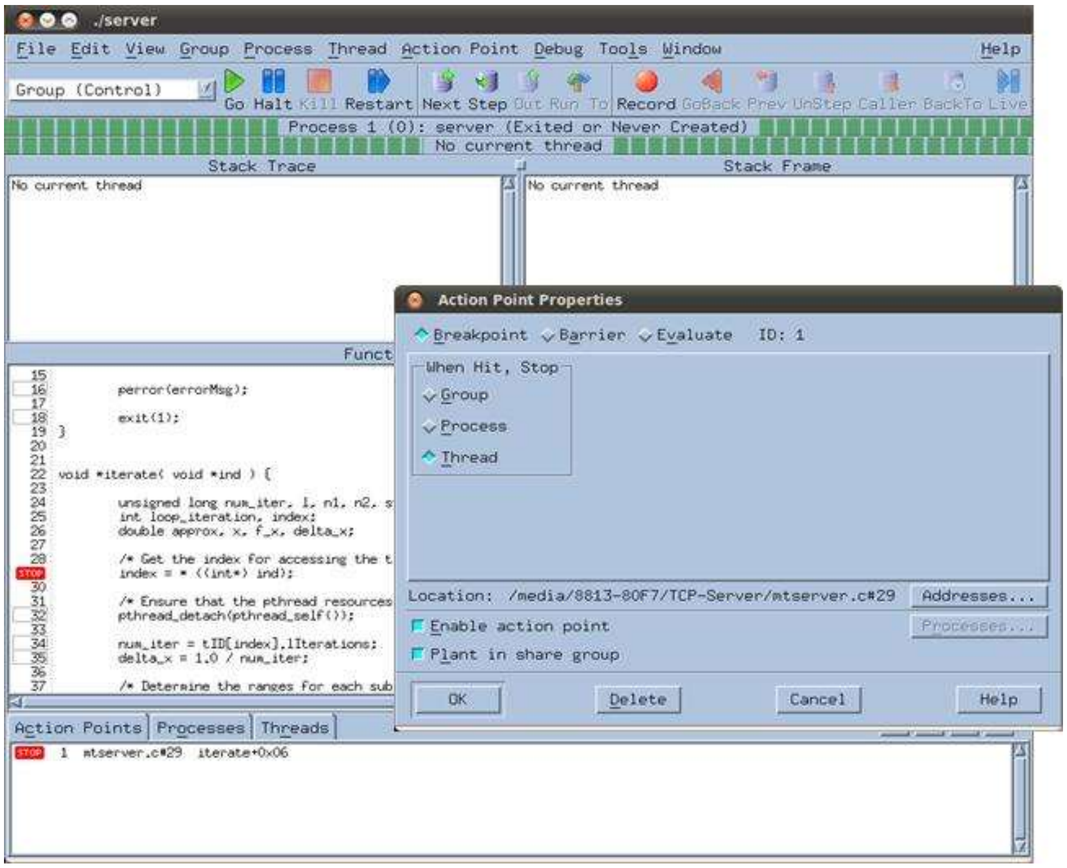
Figure 24: GDB thread 4 terminates

Here GDB does demonstrate the error, but you have to follow the lines as they come up quite carefully. In this case the difference between flow control in thread 3 and thread 4 leads thread 4’s results to be disregarded.

Thread Control Capabilities of TotalView

The following steps will show the advantages of TotalView's thread control capabilities:

1.) Start the server with the command **`totalview ./server`** and place a single breakpoint at line 29. Use the 'Action Point Properties' dialog to tell the breakpoint to act only on individual threads, as seen in Figure 25.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Stack Trace
Stack Frame
Funct
Action Points Processes Threads

```
Process 1 (0): server (Exited or Never Created)
No current thread
```

```
No current thread
```

```
15
16          perror(errorMsg);
17
18          exit(1);
19  }
20
21
22  void *iterate( void *ind ) {
23
24          unsigned long num_iter, l, n1, n2, s
25          int loop_iteration, index;
26          double approx, x, f_x, delta_x;
27
28          /* Get the index for accessing the t
STOP        index = * ((int*) ind);
30
31          /* Ensure that the pthread resources
32          pthread_detach(pthread_self());
33
34          num_iter = tID[index].lIterations;
35          delta_x = 1.0 / num_iter;
36
37          /* Determine the ranges for each sub
```

```
STOP  1  mtserver.c#29  iterate+0x06
```

Action Point Properties

[✓ Breakpoint] Barrier Evaluate ID: 1

When Hit, Stop: Group Process [✓ Thread]

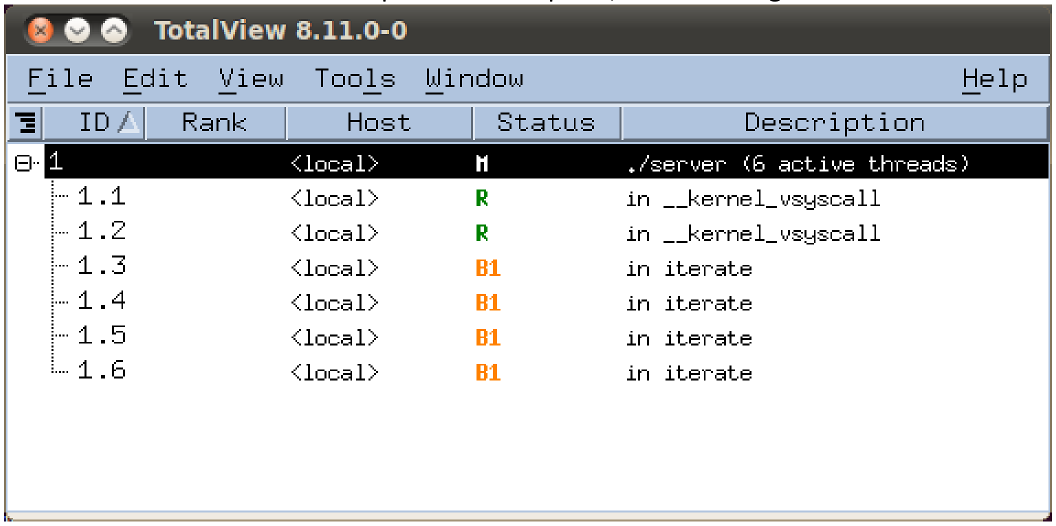
Location: /media/8813-80F7/TCP-Server/mtserver.c#29

[✓] Enable action point

[✓] Plant in share group

Figure 25: Setting Action Point properties in TotalView

2.) Run the server by pushing the ‘Go’ button and run the client one time as already described to force all four sub-threads to stop at the breakpoint, as seen in Figure 26.

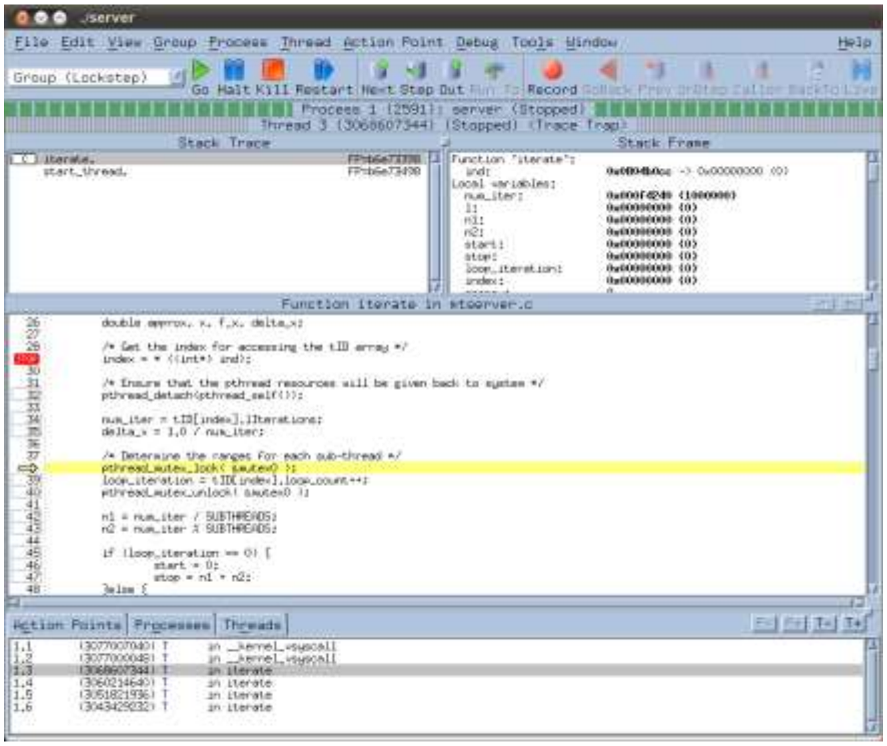


```
ID    Rank   Host      Status   Description
1            <local>   M        ./server (6 active threads)
  1.1        <local>   R        in __kernel_vsyscall
  1.2        <local>   R        in __kernel_vsyscall
  1.3        <local>   B1       in iterate
  1.4        <local>   B1       in iterate
  1.5        <local>   B1       in iterate
  1.6        <local>   B1       in iterate
```

Figure 26: TotalView Root Window showing all threads have been stopped at the breakpoint

3.) Move your cursor down to highlight one of the sub-threads.

4.) As seen in Figure 27, select the scope modifier ‘Group (Lockstep)’ in the toolbar’s P/T Set Control pull down menu to allow single-stepping of all four sub-threads in sync. It makes sense to move all threads together until line 38 because the mutex will only allow one thread at a time to pass this barrier. Single-stepping in TotalView is done by pushing either the ‘Next’ (stepping over functions) or ‘Step’ (stepping into functions) buttons.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Lockstep)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Stack Trace
Stack Frame
Action Points Processes Threads

```
Process 1 (2591): server (Stopped)
Thread 3 (3068607344) (Stopped) <Trace Trap>

iterate,                 FP=b6e73398
start_thread,            FP=b6e73498

Function "iterate":
  ind:                   0x0804b0c -> 0x00000000 (0)
Local variables:
  num_iter:              0x000f4240 (1000000)
  i:                     0x00000000 (0)
  n1:                    0x00000000 (0)
  n2:                    0x00000000 (0)
  start:                 0x00000000 (0)
  stop:                  0x00000000 (0)
  loop_iteration:        0x00000000 (0)
  index:                 0x00000000 (0)

Function iterate in mtserver.c
26        double approx, x, f_x, delta_x;
27
28        /* Get the index for accessing the tID array */
STOP      index = * ((int*) ind);
30
31        /* Ensure that the pthread resources will be given back to system */
32        pthread_detach(pthread_self());
33
34        num_iter = tID[index].iIterations;
35        delta_x = 1.0 / num_iter;
36
37        /* Determine the ranges for each sub-thread */
=>        pthread_mutex_lock( &mutex0 );
39        loop_iteration = tID[index].loop_count++;
40        pthread_mutex_unlock( &mutex0 );
41
42        n1 = num_iter / SUBTHREADS;
43        n2 = num_iter % SUBTHREADS;
44
45        if (loop_iteration == 0) {
46                start = 0;
47                stop = n1 + n2;
48        }else {

1.1    (3077007040) T    in __kernel_vsyscall
1.2    (3077000048) T    in __kernel_vsyscall
1.3    (3068607344) T    in iterate
1.4    (3060214640) T    in iterate
1.5    (3051821936) T    in iterate
1.6    (3043429232) T    in iterate
```

Figure 27: Controlling the lockstep group in TotalView

Another convenient way to control a defined number of threads using TotalView is to create a Call Graph group. As seen in Figure 28, the Call Graph is a graphical representation of the recent stack trace of specified processes and threads. It can be opened by choosing 'Call Graph' on the menu 'Tools' of the main Process Window.

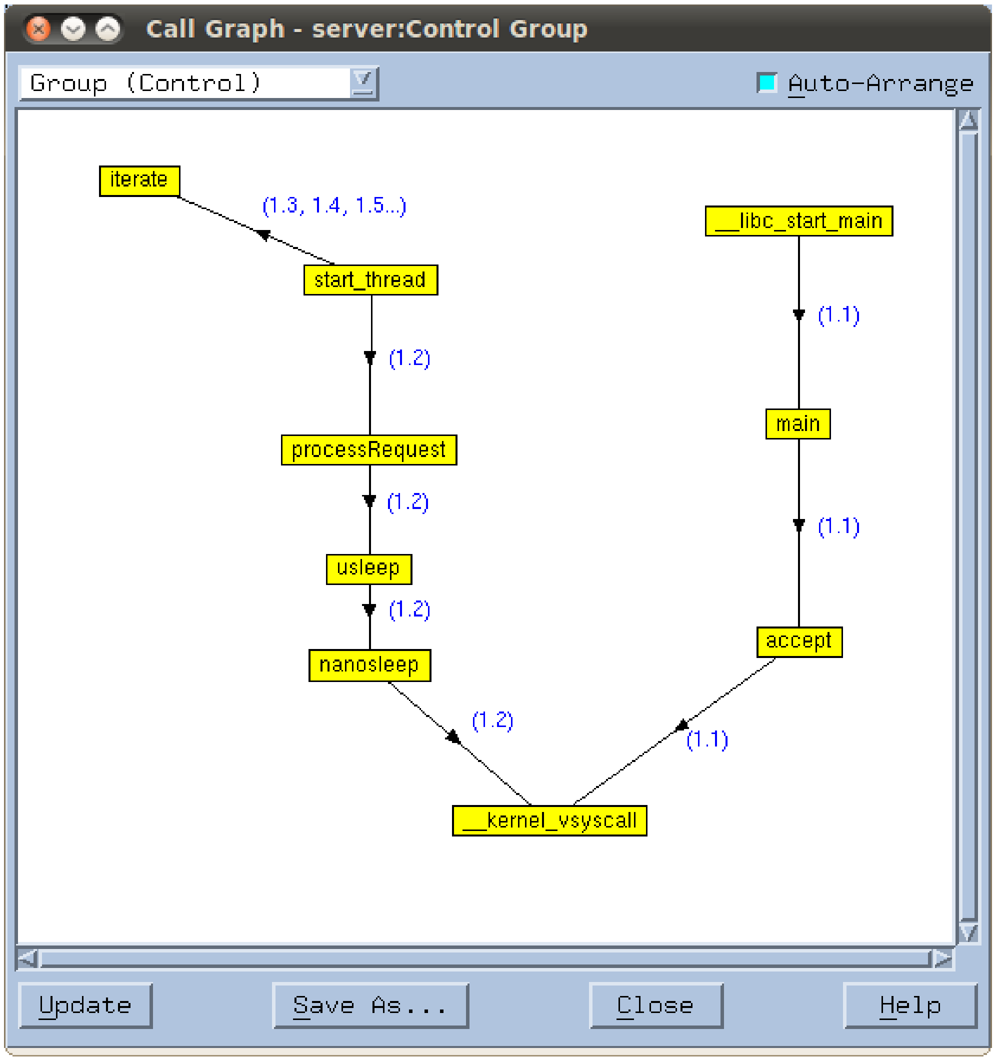


Figure 28: The TotalView Call Graph

The Call Graph is useful for understanding the functionality of the application by showing which threads have a displayed routine on their call stack. This graph shows three different thread behaviours. Thread 1.1 includes main and is currently in a routine called accept(), thread 1.2 is sleeping in a routine called processRequest(), and threads 1.3 – 1.5 are in iterate().

Additionally, double clicking on a routine in the graph automatically creates a thread group called "call_graph," containing all of the threads which call that routine, as seen in Figure 29.

The new group will immediately appear in the Process Window's pull down menu and can be selected for actions affecting the entire thread group, for example, moving this entire group of threads in single step mode to line 38.

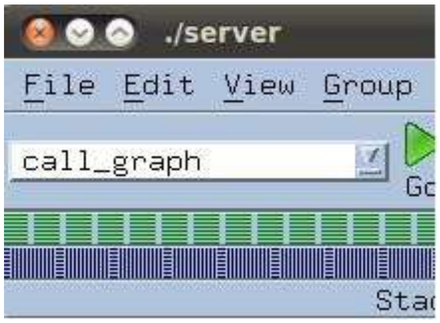
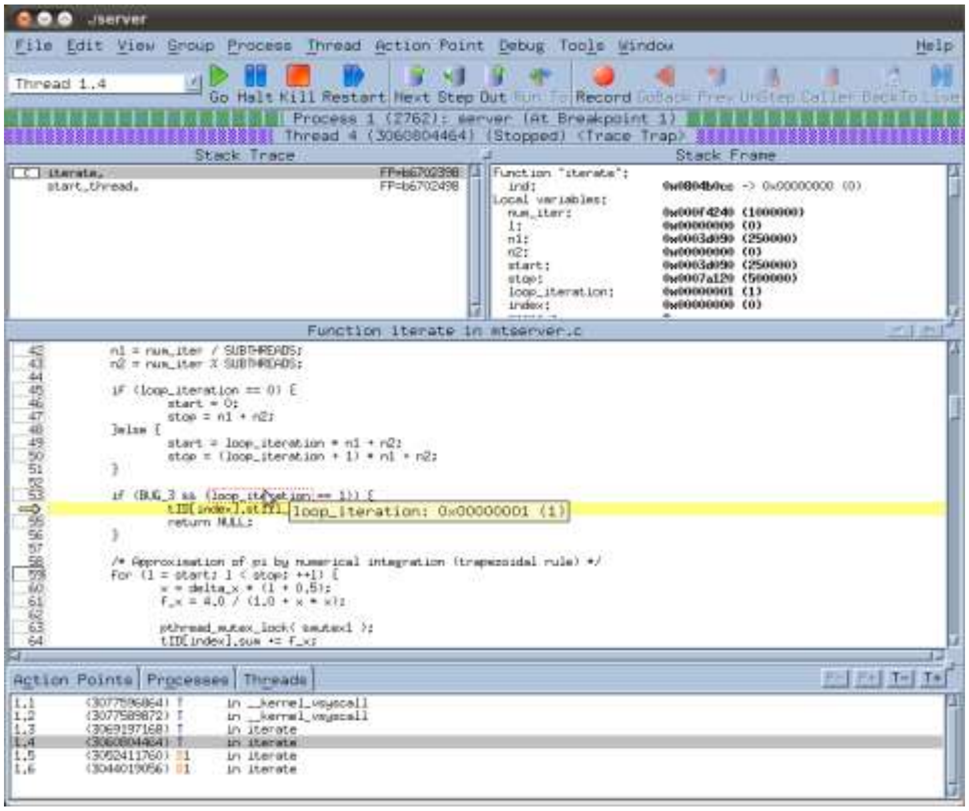


Figure 29: TotalView Process Window with the new call_graph group selected

5.) Single-step each thread starting from line 38 to easily uncover the bug, as displayed in Figure 30.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Thread 1.4
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Stack Trace
Stack Frame
Action Points Processes Threads

```
Process 1 (2762): server (At Breakpoint 1)
Thread 4 (3060804464) (Stopped) (Trace Trap)
```

```
iterate,          FP=b6702398
start_thread,     FP=b6702498
```

```
Function "iterate":
  ind:                 0x0804b0cc -> 0x00000000 (0)
Local variables:
  num_iter:            0x000f4240 (1000000)
  l:                   0x00000000 (0)
  n1:                  0x0003d090 (250000)
  n2:                  0x00000000 (0)
  start:               0x0003d090 (250000)
  stop:                0x0007a120 (500000)
  loop_iteration:      0x00000001 (1)
  index:               0x00000000 (0)
```

```
Function iterate in mtserver.c

        n1 = num_iter / SUBTHREADS;
        n2 = num_iter % SUBTHREADS;

        if (loop_iteration == 0) {
                start = 0;
                stop = n1 + n2;
        }else {
                start = loop_iteration * n1 + n2;
                stop = (loop_iteration + 1) * n1 + n2;
        }

        if (BUG_3 && (loop_iteration == 1)) {
                tID[index].st[illegible] loop_iteration: 0x00000001 (1)
                return NULL;
        }

        /* Approximation of pi by numerical integration (trapezoidal rule) */
        for (l = start; l < stop; ++l) {
               x = delta_x * (l + 0.5);
               f_x = 4.0 / (1.0 + x * x);

               pthread_mutex_lock( &mutex1 );
               tID[index].sum += f_x;
```

```
1.1     (3077596864) T     in __kernel_vsyscall
1.2     (3077589872) T     in __kernel_vsyscall
1.3     (3069197168) T     in iterate
1.4     (3060804464) T     in iterate
1.5     (3052411760) B1    in iterate
1.6     (3044019056) B1    in iterate
```

Figure 30: The bug has been found in TotalView

Stepping each thread through the section shows that one thread takes a divergent path. Hovering over the variables shows why the program is taking that path.

Compared to GDB, the TotalView debugger has far superior thread capabilities. Not only does it offer a convenient interface designed to allow complete control over individual threads, it also supports asynchronous thread control, thread groups, stepping commands that operate on thread groups, and thread width breakpoints.

Displaying Local Variables of Different Threads

It is usually necessary to monitor the local variables belonging to each thread in order to check whether a program is working as expected. In order to demonstrate such a case, rebuild the server with all of the bugs turned off and the number of sub-threads set to 10.

In this debugging session, monitor changes to the local variable ***f_x*** for each of the 10 sub-threads while in single-step mode. To accomplish this, place a breakpoint at line 60.

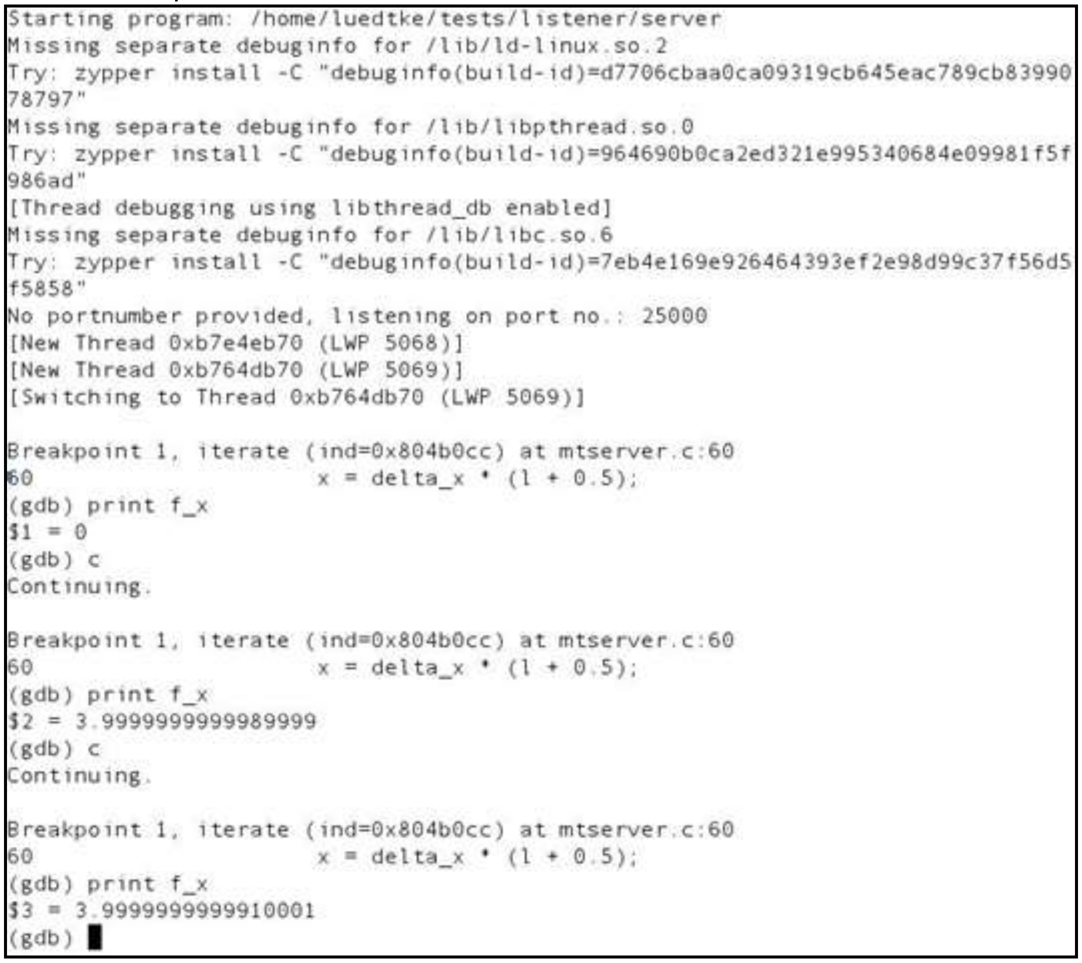
As with the sessions above, GDB will be demonstrated first.

Displaying Local Variables of Different Threads Using GDB

1. Start the server by typing **`gdb ./server <return>`** , set the breakpoint by typing **`break 60 <return>`**, run the server by typing **`run <return>`** and send a client request to the server by typing **`./client localhost 25000 1000000 <return>`** in a separate console. The first of the 10 sub-threads will hit the breakpoint and the process will stop.

2. Because GDB always focuses on the thread that last hit the breakpoint, you can serially display the value of ***f_x*** for each thread by alternating the commands **`print f_x <return>`** and **`c <return>`**.

As shown in Figure 31, the nature of the GDB output makes it very difficult to compare the values stored by the various threads, especially if hundreds or thousands of threads are running simultaneously.



```
Starting program: /home/luedtke/tests/listener/server
Missing separate debuginfo for /lib/ld-linux.so.2
Try: zypper install -C "debuginfo(build-id)=d7706cbaa0ca09319cb645eac789cb83990
78797"
Missing separate debuginfo for /lib/libpthread.so.0
Try: zypper install -C "debuginfo(build-id)=964690b0ca2ed321e995340684e09981f5f
986ad"
[Thread debugging using libthread_db enabled]
Missing separate debuginfo for /lib/libc.so.6
Try: zypper install -C "debuginfo(build-id)=7eb4e169e926464393ef2e98d99c37f56d5
f5858"
No portnumber provided, listening on port no.: 25000
[New Thread 0xb7e4eb70 (LWP 5068)]
[New Thread 0xb764db70 (LWP 5069)]
[Switching to Thread 0xb764db70 (LWP 5069)]

Breakpoint 1, iterate (ind=0x804b0cc) at mtserver.c:60
60                    x = delta_x * (1 + 0.5);
(gdb) print f_x
$1 = 0
(gdb) c
Continuing.

Breakpoint 1, iterate (ind=0x804b0cc) at mtserver.c:60
60                    x = delta_x * (1 + 0.5);
(gdb) print f_x
$2 = 3.9999999999989999
(gdb) c
Continuing.

Breakpoint 1, iterate (ind=0x804b0cc) at mtserver.c:60
60                    x = delta_x * (1 + 0.5);
(gdb) print f_x
$3 = 3.9999999999910001
(gdb) █
```

Figure 31: GDB displaying the value of a local variable for every thread

Of course, it might be nice for GDB to tell you which thread you are seeing each data element from; you can inquire into it with info threads.

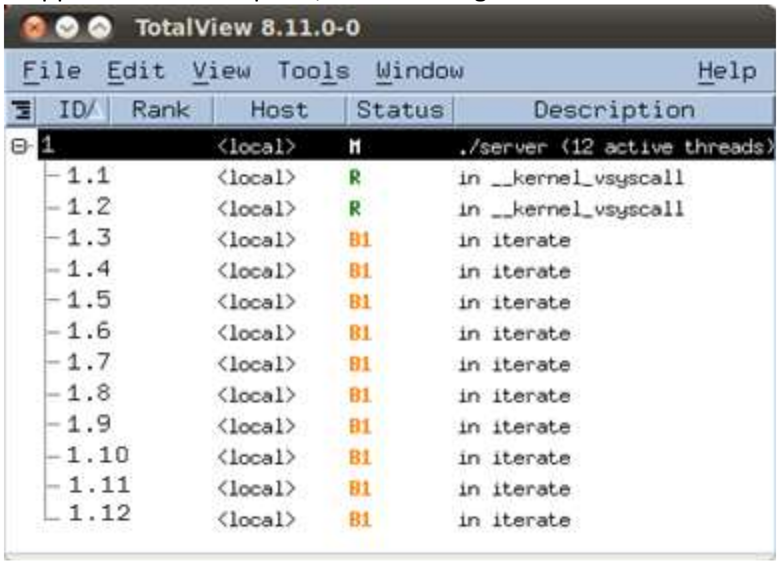
Displaying Local Variables of Different Threads Using TotalView

The TotalView session is set-up similar to the GDB session above.

1.) Start the server within TotalView by typing **`totalview ./server <return>`** and set a breakpoint in front of line 60 by clicking on the rectangle around the line number with the left mouse button.

2.) Using the right mouse button, click on the breakpoint to bring up its properties menu. Set the breakpoint to act on threads only.

3.) Start the server by pushing the 'Go' button and send it a client request by typing **`./client localhost 25000 1000000 <return>`** in a separate console. The 10 sub-threads stopped at the breakpoint, as seen in Figure 32.



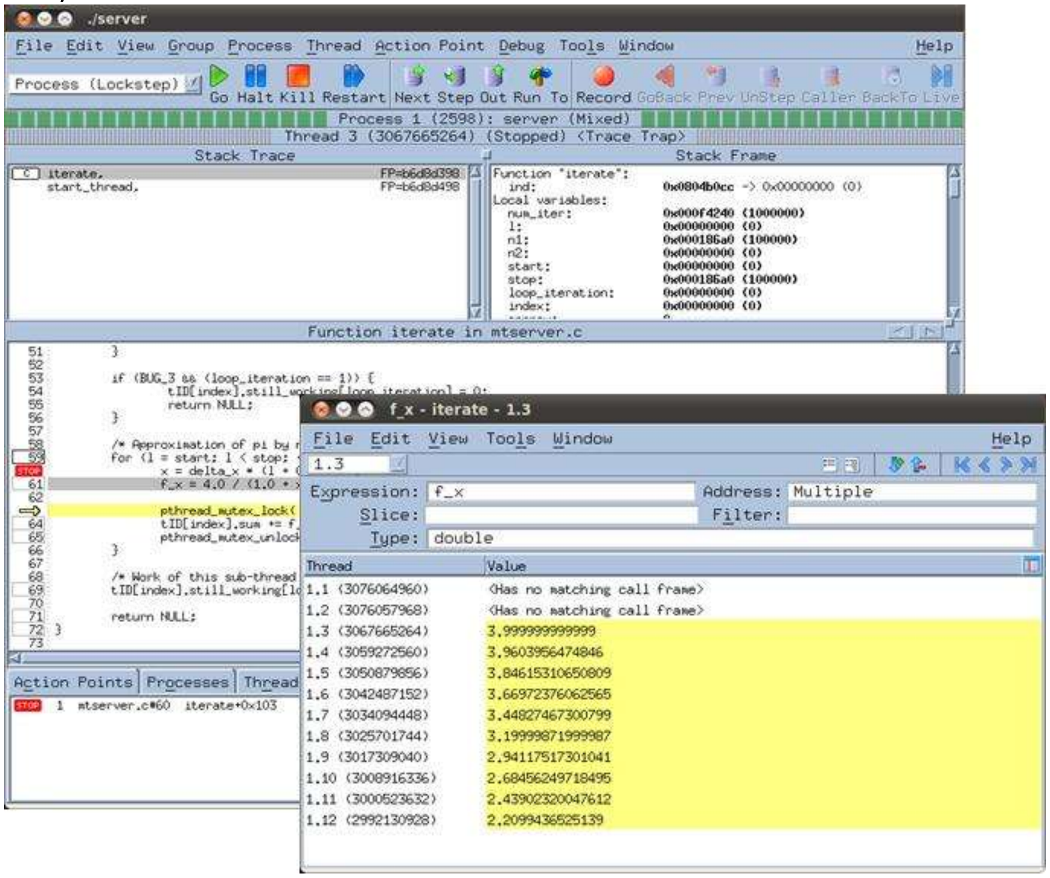
ID	Rank	Host	Status	Description
1		<local>	M	./server (12 active threads)
1.1		<local>	R	in __kernel_vsyscall
1.2		<local>	R	in __kernel_vsyscall
1.3		<local>	B1	in iterate
1.4		<local>	B1	in iterate
1.5		<local>	B1	in iterate
1.6		<local>	B1	in iterate
1.7		<local>	B1	in iterate
1.8		<local>	B1	in iterate
1.9		<local>	B1	in iterate
1.10		<local>	B1	in iterate
1.11		<local>	B1	in iterate
1.12		<local>	B1	in iterate

Figure 32: The TotalView Root Window showing all sub-threads stopped at breakpoint 1

4.) In the Root Window, double-click on the line marked 1.3 to change the focus to thread 3. The current position of thread 3 is highlighted in the Process Window's source code pane.

5.) Drill down into the variable ***f_x*** by double-clicking it in the source code pane. This opens a variable window that displays the value of ***f_x*** for the currently selected thread, in this case thread 3.

6.) The ‘View’ menu of the variable window includes the option ‘Show Across ->, as seen in Figure 33.’ Select ‘Show Across ->Threads’ to create an array-like view of ***f_x***, showing the values for every thread.


./server
File Edit View Group Process Thread Action Point Debug Tools Window Help
Process (Lockstep)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Stack Trace
Stack Frame
Function iterate in mtserver.c
Action Points Processes Thread
f_x - iterate - 1.3
File Edit View Tools Window Help
1.3

```
Process 1 (2598): server (Mixed)
Thread 3 (3067665264) (Stopped) <Trace Trap>
```

```
C  iterate,          FP=b6d8d398
   start_thread,     FP=b6d8d498
```

```
Function "iterate":
  ind:                 0x0804b0cc -> 0x00000000 (0)
Local variables:
  num_iter:            0x000f4240 (1000000)
  l:                   0x00000000 (0)
  n1:                  0x000186a0 (100000)
  n2:                  0x00000000 (0)
  start:               0x00000000 (0)
  stop:                0x000186a0 (100000)
  loop_iteration:      0x00000000 (0)
  index:               0x00000000 (0)
```

```
        }

        if (BUG_3 && (loop_iteration == 1)) {
                tID[index].still_working[loop_iteration] = 0;
                return NULL;
        }

        /* Approximation of pi by r
        for (l = start; l < stop; 
                x = delta_x * (l + 0
                f_x = 4.0 / (1.0 + x

                pthread_mutex_lock(
                tID[index].sum += f_
                pthread_mutex_unlock
        }

        /* Work of this sub-thread
        tID[index].still_working[lo

        return NULL;
}
```

```
STOP  1  mtserver.c#60  iterate+0x103
```

```
Expression: f_x         Address: Multiple
     Slice:              Filter:
      Type: double
```

Thread	Value
1.1 (3076064960)	<Has no matching call frame>
1.2 (3076057968)	<Has no matching call frame>
1.3 (3067665264)	3.999999999999
1.4 (3059272560)	3.9603956474846
1.5 (3050879856)	3.84615310650809
1.6 (3042487152)	3.66972376062565
1.7 (3034094448)	3.44827467300799
1.8 (3025701744)	3.19999871999987
1.9 (3017309040)	2.94117517301041
1.10 (3008916336)	2.68456249718495
1.11 (3000523632)	2.43902320047612
1.12 (2992130928)	2.2099436525139

Figure 33: TotalView’s Show Across capability

Variables displayed in a variable window are updated and highlighted as changes to them occur.

View across makes it much easier to do thread vs. thread data comparisons. With TotalView you can synchronize your threads so they are all at the same well-understood point in their execution, and do an apples-to-apples comparison of the data each thread contains at that same point. Alternately, you can run the threads in any execution sequence and see what happens, not just in that thread, but in all the other threads.

Conclusion

This paper shows what it is like to debug several common problems in a multithreaded application using basic scenarios. It demonstrates how this debugging can be done with the open source GDB and with the TotalView debugger. TotalView provides a variety of useful features that make it easy to see and control what is going on in a multithreaded program. As you debug and explore the behavior of a multithreaded application, features such as asynchronous thread control, thread groups, thread width breakpoints, the Call Graph display, the ability to easily navigate data, and the ability to view variables across multiple threads will help to provide a clear indication of what the threads are doing.

Multithreaded programs can manifest intermittent defects. Tracking and resolving these defects without the right tools can be a tedious and frustrating process of running and re-running the program, only to have the problem go away. TotalView gives developers the ability to capture and deterministically replay the execution history of the program. This radically simplifies the troubleshooting process. Once an intermittent error is captured in the recording, it is easy to go back and forth as many times as needed to understand what has happened, making errors easier to solve.

About TotalView

TotalView is a scalable and intuitive debugger for parallel applications written in C, C++, and Fortran. Designed to improve developer productivity, TotalView simplifies and shortens the process of developing, debugging, and optimizing complex applications. TotalView provides a powerful combination of capabilities for pinpointing and fixing hard-to-find bugs, such as race conditions, memory leaks, and memory overruns. Providing developers the ability to step freely, both forwards and backwards, through program execution, TotalView's unique reverse debugging capabilities drastically reduce the amount of time invested in troubleshooting code. To help developers maximize hardware capabilities, TotalView also provides debugging support for NVIDIA® CUDA™, OpenACC®, and the Intel® Xeon® Phi™ coprocessor.

About Rogue Wave Software