# Multi-threaded Debugging Using the GNU Project Debugger


## Abstract

Debugging is a fundamental step of software development and increasingly complex software requires greater specialization from debugging tools. Software built with multiple threads of execution in particular requires special considerations and comprehensive debugging management. The free, open source GNU Project Debugger has the capacity to observe all threads while the program is running and then isolate a current thread to show focused information for debugging. This article explores some of the GDB's features for multi-threaded software debugging and includes code examples demonstrating setup of thread management and execution of debugging procedures.


## Introduction

This article presents the basic functionalities of the GNU Project Debugger. It will introduce the specific vocabulary and commands used by the GDB for managing multi-threaded application testing. The GDB will run on Unix and Microsoft Windows OSs and debug programs written in Ada, C, C++, Objective-C, Pascal, and many other languages. By the end of this article, a developer should have a number of new techniques to try right away and a solid foundation for approaching more complicated multi-threaded debugging projects.

The GNU Project Debugger and user manuals are available for free download from www.gnu.org/software/gdb/, along with links to product documentation and a FAQ wiki.

## Multi-threaded Debugging Using GDB

For POSIX threads, debugging is generally accomplished using the GNU Project Debugger (GDB). GDB provides a number of capabilities for debugging threads, including:

- Automatic notification when new threads are created
- Listing of all threads in the system
- Thread-specific breakpoints
- The ability to switch between threads
- The ability to apply commands to a group of threads

Not all GDB implementations support all of the features outlined here. Please refer to your system's manual pages for a complete list of supported features.

## Notification on Thread Creation

When GDB detects that a new thread is created, it displays a message specifying the thread's identification on the current system. This identification, known as the *systag*, varies from platform to platform.

Here is an example of this notification:

```
Starting program: /home/user/threads
    [Thread debugging using libthread_db enabled]
    [New Thread -151132480 (LWP 4445)]
    [New Thread -151135312 (LWP 4446)]
```

Keep in mind that the systag is the operating system's identification for a thread, not GDB's. GDB assigns each thread a unique number that identifies it for debugging purposes.

## Getting a List of All Threads in the Application

GDB provides the generic info command to get a wide variety of information about the program being debugged. It is no surprise that a subcommand of info would be *info threads*. This command prints a list of threads running in the system:

```
(gdb) info threads
2 Thread -151135312 (LWP 4448) 0x00905f80 in vfprintf () from /lib/tls/libc.so.6
* 1 Thread -151132480 (LWP 4447) main () at threads.c:27
```

The info threads command displays a table that lists three properties of the threads in the system: the thread number attached to the thread by GDB, the systag value, and the current stack frame for the current thread. The currently active thread is denoted by GDB with the * symbol. The thread number is used in all other commands in GDB.

## Setting Thread-specific Breakpoints

GDB allows users that are debugging multi-threaded applications to choose whether or not to set a breakpoint on all threads or on a particular thread. The much like the info command, this capability is enabled via an extended parameter that's specified in the break command. The general form of this instruction is:

```
break linespec thread threadnum
```

where *linespec* is the standard gdb syntax for specifying a breakpoint, and *threadnum* is the thread number obtained from the info threads command. If the thread *threadnum* arguments are omitted, the

breakpoint applies to all threads in your program. Thread-specific breakpoints can be combined with conditional breakpoints:

```
(gdb) break buffer.c:33 thread 7 if level > watermark
```

Note that stopping on a breakpoint stops all threads in your program. Generally speaking this is a desirable effect—it allows a developer to examine the entire state of an application, and the ability to switch the current thread. These are good things.

Developers should keep certain behaviors in mind, however, when using breakpoints from within GDB. The first issue is related to how system calls behave when they are interrupted by the debugger. To illustrate this point, consider a system with two threads. The first thread is in the middle of a system call when the second thread reaches a breakpoint. When the breakpoint is triggered, the system call may return early. The reason—GDB uses signals to manage breakpoints. The signal may cause a system call to return prematurely. To illustrate this point, let's say that thread 1 was executing the system call `sleep(30)`. When the breakpoint in thread 2 is hit, the sleep call will return, regardless of how long the thread has actually slept. To avoid unexpected behavior due to system calls returning prematurely, it is advisable that you check the return values of all system calls and handle this case. In this example, `sleep()` returns the number of seconds left to sleep. This call can be placed inside of a loop to guarantee that the sleep has occurred for the amount of time specified. This is shown in Listing 1.1.

```
int sleep_duration = 30;
do
{
        sleep_duration = sleep(sleep_duration);
} while (sleep_duration > 0);
```

**Listing 1.1** Proper Error Handling of System Calls

The second point to keep in mind is that GDB does not single step all threads in lockstep. Therefore, when single-stepping a line of code in one thread, you may end up executing a lot of code in other threads prior to returning to the thread that you are debugging. If you have breakpoints in other threads, you may suddenly jump to those code sections. On some OSs, GDB supports a scheduler locking mode via the set scheduler-locking command. This allows a developer to specify that the current thread is the only thread that should be allowed to run.

## Switching between Threads

In GDB, the *thread* command may be used to switch between threads. It takes a single parameter, the thread number returned by the info threads command. Here is an example of the thread command:

```
(gdb) thread 2
[Switching to thread 2 (Thread -151135312 (LWP 4549))]#0
PrintThreads (num=0xf6fddbb0) at threads.c:39
39      {
(gdb) info threads
* 2 Thread -151135312 (LWP 4549) PrintThreads (num=0xf6fddbb0) at threads.c:39
1 Thread -151132480 (LWP 4548) main () at threads.c:27
(gdb)
```

In this example, the thread command makes thread number 2 the active thread.

## Applying a Command to a Group of Threads

The thread command supports a single subcommand *apply* that can be used to apply a command to one or more threads in the application. The thread numbers can be supplied individually, or the special keyword all may be used to apply the command to all threads in the process, as illustrated in the following example:

```
(gdb) thread apply all bt

Thread 2 (Thread -151135312 (LWP 4549)):
#0 PrintThreads (num=0xf6fddbb0) at threads.c:39
#1 0x00b001d5 in start_thread () from /lib/tls/libpthread.so.0
#2 0x009912da in clone () from /lib/tls/libc.so.6

Thread 1 (Thread -151132480 (LWP 4548)):
#0 main () at threads.c:27
39      {
(gdb)
```

The GDB backtrace *(bt)* command is applied to all threads in the system. In this scenario, this command is functionally equivalent to: thread apply 2 1 bt.

## Conclusion

This article introduced debugging for multi-threaded applications via the GNU Project Debugger. Proper software engineering principles need to be followed when writing and developing robust multi-threaded applications, and that includes comprehensive testing. The GDB supports developers' need to monitor, log, and test individual threads within the program as well as overall performance. Variable execution order is usually one of the features of multi-threaded programs that makes linear or traditional debugging algorithms impossible, and the GDB neatly sidesteps this issue. However, isolating the threads and debugging them individually means that overall runtime errors may not be identified properly. For advanced debugging, consider using the Intel software tools, specifically, the Intel Debugger, the Intel Thread Checker, and the Intel Thread Profiler. The book from which this article was sampled also supplies more information on the subject, and serves as an excellent starting point for developers interested in learning more about multi-core processing and software optimization.

---

This article is based on material found in the book *Multi-Core Programming: Increasing Performance through Software Multi-threading* by Shameem Akhter and Jason Roberts. Visit the Intel Press website to learn more about this book:

http://noggin.intel.com/intelpress/categories/books/multi-core-programming

Also see our Recommended Reading List for similar topics:

http://noggin.intel.com/rr

## About the Authors

**Shameem Akhter** is a platform architect at Intel, focusing on single socket multi-core architecture and performance analysis. He has also worked as a senior software engineer with the Intel Software and Solutions Group, designing application optimizations for desktop and server platforms. Shameem holds a patent on a threading interface for constraint programming, developed as a part of his master's thesis in computer science.

**Jason Roberts** is a senior software engineer at Intel Corporation. Over the past 10 years, Jason has worked on a number of different multi-threaded software products that span a wide range of applications targeting desktop, handheld, and embedded DSP platforms.

---