Next: Forks, Previous: Inferiors and Programs, Up: Running [Contents] [Index]

4.10 Debugging Programs with Multiple Threads

In some operating systems, such as GNU/Linux and Solaris, 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 perhaps private memory.

GDB provides these facilities for debugging multi-thread programs:

- automatic notification of new threads
- ‘`thread` *thread-id*’, a command to switch among threads
- ‘`info threads`’, a command to inquire about existing threads
- ‘`thread apply [`*thread-id-list*`] [`*all*`]` *args*’, a command to apply a command to a list of threads
- thread-specific breakpoints
- ‘`set print thread-events`’, which controls printing of messages on thread start and exit.
- ‘`set libthread-db-search-path` *path*’, which lets the user specify which `libthread_db` to use if the default choice isn’t compatible with the program.

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*`]`’, where *systag* is a thread identifier whose form varies depending on the particular system. For example, on GNU/Linux, you might see

```
[New Thread 0x41e02940 (LWP 25582)]
```

when GDB notices a new thread. In contrast, on other systems, the *systag* is simply something like ‘`process 368`’, with no further qualifier.

For debugging purposes, GDB associates its own thread number —always a single integer—with each thread of an inferior. This number is unique between all threads of an inferior, but not unique between threads of different inferiors.

You can refer to a given thread in an inferior using the qualified *inferior-num.thread-num* syntax, also known as *qualified thread ID*, with *inferior-num*

being the inferior number and *thread-num* being the thread number of the given inferior. For example, thread `2.3` refers to thread number 3 of inferior 2. If you omit *inferior-num* (e.g., `thread 3`), then GDB infers you're referring to a thread of the current inferior.

Until you create a second inferior, GDB does not show the *inferior-num* part of thread IDs, even though you can always use the full *inferior-num.thread-num* form to refer to threads of inferior 1, the initial inferior.

Some commands accept a space-separated *thread ID list* as argument. A list element can be:

1. A thread ID as shown in the first field of the '`info threads`' display, with or without an inferior qualifier. E.g., '`2.1`' or '`1`'.
2. A range of thread numbers, again with or without an inferior qualifier, as in *inf.thr1-thr2* or *thr1-thr2*. E.g., '`1.2-4`' or '`2-4`'.
3. All threads of an inferior, specified with a star wildcard, with or without an inferior qualifier, as in *inf.*`*` (e.g., '`1.*`') or `*`. The former refers to all threads of the given inferior, and the latter form without an inferior qualifier refers to all threads of the current inferior.

For example, if the current inferior is 1, and inferior 7 has one thread with ID 7.1, the thread list '`1 2-3 4.5 6.7-9 7.*`' includes threads 1 to 3 of inferior 1, thread 5 of inferior 4, threads 7 to 9 of inferior 6 and all threads of inferior 7. That is, in expanded qualified form, the same as '`1.1 1.2 1.3 4.5 6.7 6.8 6.9 7.1`'.

In addition to a *per-inferior* number, each thread is also assigned a unique *global* number, also known as *global thread ID*, a single integer. Unlike the thread number component of the thread ID, no two threads have the same global ID, even when you're debugging multiple inferiors.

From GDB's perspective, a process always has at least one thread. In other words, GDB assigns a thread number to the program's "main thread" even if the program is not multi-threaded.

The debugger convenience variables '`$_thread`' and '`$_gthread`' contain, respectively, the per-inferior thread number and the global thread number of the current thread. You may find this useful in writing breakpoint conditional expressions, command scripts, and so forth. See Convenience Variables, for general information on convenience variables.

If GDB detects the program is multi-threaded, it augments the usual message about stopping at a breakpoint with the ID and name of the thread that hit the breakpoint.

```
Thread 2 "client" hit Breakpoint 1, send_message () at client.c:68
```

Likewise when the program receives a signal:

```
Thread 1 "main" received signal SIGINT, Interrupt.
```

`info threads [thread-id-list]`

Display information about one or more threads. With no arguments displays information about all threads. You can specify the list of threads that you want to display using the thread ID list syntax (see thread ID lists).

GDB displays for each thread (in this order):

1. the per-inferior thread number assigned by GDB
2. the global thread number assigned by GDB, if the '`-gid`' option was specified
3. the target system's thread identifier (*systag*)
4. the thread's name, if one is known. A thread can either be named by the user (see `thread name`, below), or, in some cases, by the program itself.
5. 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
  Id   Target Id         Frame
* 1    process 35 thread 13  main (argc=1, argv=0x7ffffff8)
  2    process 35 thread 23  0x34e5 in sigpause ()
  3    process 35 thread 27  0x34e5 in sigpause ()
    at threadtest.c:68
```

If you're debugging multiple inferiors, GDB displays thread IDs using the qualified *inferior-num.thread-num* format. Otherwise, only *thread-num* is shown.

If you specify the '`-gid`' option, GDB displays a column indicating each thread's global thread ID:

```
(gdb) info threads
  Id   GId  Target Id             Frame
  1.1  1    process 35 thread 13  main (argc=1, argv=0x7ffffff8)
  1.2  3    process 35 thread 23  0x34e5 in sigpause ()
  1.3  4    process 35 thread 27  0x34e5 in sigpause ()
* 2.1  2    process 65 thread 1   main (argc=1, argv=0x7ffffff8)
```

On Solaris, you can display more information about user threads with a Solaris-specific command:

`maint info sol-threads`

Display info on Solaris user threads.

`thread thread-id`

Make thread ID *thread-id* the current thread. The command argument *thread-id* is the GDB thread ID, as shown in the first field of the '`info`

`threads`' display, with or without an inferior qualifier (e.g., '`2.1`' or '`1`').

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 (Thread 0xb7fdab70 (LWP 12747))]
#0  some_function (ignore=0x0) at example.c:8
8           printf ("hello\n");
```

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

`thread apply [`*`thread-id-list`*` | all [-ascending]] `*`command`*

The `thread apply` command allows you to apply the named *command* to one or more threads. Specify the threads that you want affected using the thread ID list syntax (see thread ID lists), or specify `all` to apply to all threads. To apply a command to all threads in descending order, type `thread apply all `*`command`*. To apply a command to all threads in ascending order, type `thread apply all -ascending `*`command`*.

`thread name [`*`name`*`]`

This command assigns a name to the current thread. If no argument is given, any existing user-specified name is removed. The thread name appears in the '`info threads`' display.

On some systems, such as GNU/Linux, GDB is able to determine the name of the thread as given by the OS. On these systems, a name specified with '`thread name`' will override the system-give name, and removing the user-specified name will cause GDB to once again display the system-specified name.

`thread find [`*`regexp`*`]`

Search for and display thread ids whose name or *systag* matches the supplied regular expression.

As well as being the complement to the '`thread name`' command, this command also allows you to identify a thread by its target *systag*. For instance, on GNU/Linux, the target *systag* is the LWP id.

```
(GDB) thread find 26688
Thread 4 has target id 'Thread 0x41e02940 (LWP 26688)'
(GDB) info thread 4
  Id   Target Id         Frame
  4    Thread 0x41e02940 (LWP 26688) 0x00000031ca6cd372 in select ()
```

`set print thread-events`

`set print thread-events on`

`set print thread-events off`

The `set print thread-events` command allows you to enable or disable

printing of messages when GDB notices that new threads have started or that threads have exited. By default, these messages will be printed if detection of these events is supported by the target. Note that these messages cannot be disabled on all targets.

`show print thread-events`

Show whether messages will be printed when GDB detects that threads have started and exited.

See Stopping and Starting Multi-thread Programs, for more information about how GDB behaves when you stop and start programs with multiple threads.

See Setting Watchpoints, for information about watchpoints in programs with multiple threads.

`set libthread-db-search-path [path]`

If this variable is set, *path* is a colon-separated list of directories GDB will use to search for `libthread_db`. If you omit *path*, '`libthread-db-search-path`' will be reset to its default value (`$sdir:$pdir` on GNU/Linux and Solaris systems). Internally, the default value comes from the `LIBTHREAD_DB_SEARCH_PATH` macro.

On GNU/Linux and Solaris systems, GDB uses a "helper" `libthread_db` library to obtain information about threads in the inferior process. GDB will use '`libthread-db-search-path`' to find `libthread_db`. GDB also consults first if inferior specific thread debugging library loading is enabled by '`set auto-load libthread-db`' (see libthread_db.so.1 file).

A special entry '`$sdir`' for '`libthread-db-search-path`' refers to the default system directories that are normally searched for loading shared libraries. The '`$sdir`' entry is the only kind not needing to be enabled by '`set auto-load libthread-db`' (see libthread_db.so.1 file).

A special entry '`$pdir`' for '`libthread-db-search-path`' refers to the directory from which `libpthread` was loaded in the inferior process.

For any `libthread_db` library GDB finds in above directories, GDB attempts to initialize it with the current inferior process. If this initialization fails (which could happen because of a version mismatch between `libthread_db` and `libpthread`), GDB will unload `libthread_db`, and continue with the next directory. If none of `libthread_db` libraries initialize successfully, GDB will issue a warning and thread debugging will be disabled.

Setting `libthread-db-search-path` is currently implemented only on some platforms.

`show libthread-db-search-path`

Display current libthread_db search path.

```
set debug libthread-db
show debug libthread-db
```

Turns on or off display of `libthread_db`-related events. Use `1` to enable, `0` to disable.

Next: Forks, Previous: Inferiors and Programs, Up: Running [Contents] [Index]