Debugging Embedded Linux Platforms with GDB and Python

Give your debugging sessions go-faster stripes with the power of Python.

TOM PARKIN

If you write code for Linux systems, chances are you will have used the venerable GNU Debugger (GDB). Acting as a back end for many GUIs and the interface to various JTAG debugging tools in the embedded world, GDB is the foremost debugger for Linux. As of release 7.0, GDB gained a compelling new capability: support for scripting debugging actions using a Python interpreter. In this article, I take a look at how to drive GDB using Python and apply this knowledge to the vexatious issue of debugging an embedded Linux platform.

The Challenge of Embedded Linux Debugging

Debugging Linux programs on an x86 PC platform, although not necessarily easy, at least is well supported by a variety of tools. Most Linux distributions package development and debugging tools to assist with anything from profiling runtime performance through tracing memory leaks and detecting deadlocks.

Embedded platforms are rarely so well served. Although a number of projects seek to provide the kind of polish and integration for embedded development that is taken for granted on the desktop, these are not yet widely adopted in all areas of embedded Linux development. Many embedded devices are developed using what effectively is a handcrafted Linux distribution, closely tied to the specific goals of that device. The time required to integrate a wide range of handy debugging tools into that environment, especially in the fast-paced world of consumer electronics, is an overhead few teams can meet.

Many embedded platforms seek to save on resource overhead through the use of "low-fat" system libraries (such as uClibc in the place of glibc), which may make the integration of some debugging tools more difficult. Indeed, in some cases, the architecture of the CPU used by the target platform will prevent the use of certain tools altogether. The excellent Valgrind instrumentation framework, for example, has limited support for the ARM architecture and no support at all for MIPS or SuperH.

Happily, GDB is widely available for embedded devices because it is easy to cross-compile and supports a wide range of target architectures.

The nature of embedded devices often means that CPU cycles and memory are scarce. Any debugging tool that weighs too heavily on either may make its use impractical on an embedded device, especially when attempting to debug race conditions and the like.

The net result of this inconsistent provision of debugging tools across the embedded Linux world is that most developers have to makeshift as best they can with the tools that are available. Happily, GDB is widely available for embedded devices because it is easy to cross-compile and supports a wide range of target architectures. And with the recent integration of Python scripting support, GDB can extended beyond the typical debugging tasks of single stepping and variable examination.

Scripting GDB with Python

GDB has long supported extension via pre-canned sequences of debugger commands. This ability makes it possible to automate certain parts of a debugging work flow and even implement new debugger functions.

Integrating Python into GDB adds an extra dimension to the possibilities of GDB scripting and extension. In addition to the simple functions and flow control of GDB's native scripting language, the full power of the Python language is made available.

The Python GDB API is presented as a Python module called gdb, which provides access to GDB's internal representation of a process under debugging. The module includes interfaces to process information, threads, stack frames, values and types, functions, symbols and break points. In addition, a mechanism is provided to inject commands into the GDB command-line interface.

The result is that the internals of GDB are now available as a rich set of libraries for programmatic driving of the debugger. This creates a whole range of new opportunities for extension and automation. For example, let's imagine you want to debug calls to malloc() in a large application, but you're really interested only in calls from a certain module. Ideally, you want to be able to break execution only when one of the module's functions is in the backtrace at the point that malloc() is called. The Python API gives you that flexibility.

Problem Code

To explore the use of Python within GDB, let's debug a small C program, the code for which is shown in Listing 1. It performs the simple task of printing the phrase "Hello World!" in a rather convoluted manner, and it has at least one obvious bug. Besides being over-engineered for the task at hand, hello_world.c makes use of two mutexes for serializing access to different data structures, and not all users of these mutexes agree on the order in which locks should be acquired. This quickly yields a runtime deadlock.

Although hello_world.c is somewhat contrived, it does demonstrate the kinds of runtime bugs that multithreaded applications can come across when mutexes are required to protect data structures from different contexts.

Before reading any further, it is worth considering how you might debug such a deadlock. On an x86 platform, you could consider using the Valgrind framework's drd tool. Alternatively, you may choose to recompile the code with different options to change the behavior. But what would you do if Valgrind did not work on your platform, or if the code you wanted to rebuild was a third-party library for which you had only binaries?

Setting Up the Environment: the Embedded Platform

The example platform for this article uses a little-endian MIPS-based System On a Chip (SOC) device. MIPS is widely used in home routers, such as the popular Linksys WRT54G series, as well as in many set-top box platforms for accessing digital television services. Our platform has a fairly powerful 400MHz CPU, as well as 512MB of DDR RAM, making it a quite capable embedded device. We can communicate with the platform over a serial console and using an Ethernet port.

On the software side, our platform runs a 2.6 series Linux kernel that has been extended by the SOC manufacturer to support the specific CPU we are using. It has a fairly typical userspace based around uClibc and BusyBox, along with a range of GNU utilities, such as awk and sed.

Setting Up the Environment: Cross-Compiling GDB

In order to run GDB on our embedded platform, we will make use

Listing 1. C Source Code for hello_world

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

#define THREAD_COUNT         32

/* String output data */
static const char    *string = "Hello World!\n";
static int           cursor = 0;
pthread_mutex_t      print_lock = PTHREAD_MUTEX_INITIALIZER;

/* Runtime statistics */
static int           chars_printed = 0;
pthread_mutex_t      statistics_lock = PTHREAD_MUTEX_INITIALIZER;

/* Print one character of the string "Hello World!" to stdout */
/* Returns a pointer to the character printed */
static char *say_hello(void)
{
    char *printed_letter = NULL;

    printf("%c", string[cursor]);
    if (++cursor > strlen(string)) {
        cursor = 0;
        fflush(stdout);
    }

    printed_letter = (char *) malloc(1);
    if (printed_letter) {
        *printed_letter = string[cursor];
    }

    return printed_letter;
}

/* A "bug-free" printer function */
static void *good_printer(void *data)
{
    char *c = NULL;

    while(1) {
        c = NULL;
        pthread_mutex_lock(&print_lock);
        pthread_mutex_lock(&statistics_lock);
        c = say_hello();
        if (c) free(c);
        chars_printed++;
        pthread_mutex_unlock(&statistics_lock);
        pthread_mutex_unlock(&print_lock);
    }
    return NULL;
}

/* A buggy printer function */
static void *bad_printer(void *data)
{
    while(1) {
        pthread_mutex_lock(&statistics_lock);
        pthread_mutex_lock(&print_lock);
        say_hello();
        chars_printed++;
        pthread_mutex_unlock(&print_lock);
        pthread_mutex_unlock(&statistics_lock);
    }
    return NULL;
}

int main (int argc, char **argv)
{
    pthread_t threads[THREAD_COUNT];
    int i;

    /* Spawn many good children threads */
    for (i = 1; i < THREAD_COUNT; i++) {
        if (0 != pthread_create(&threads[i], NULL, good_printer, NULL)) {
            perror("pthread_create");
            exit(EXIT_FAILURE);
        }
    }

    /* Spawn one bad child thread */
    if (0 != pthread_create(&threads[0], NULL, bad_printer, NULL)) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
    }

    pthread_join(threads[0], NULL);

    return EXIT_SUCCESS;
}
```

of the gdbserver tool for remote debugging. This allows us to run GDB on a Linux PC, connecting to the embedded target using Ethernet. The protocol GDB uses to communicate with gdbserver is compatible across releases, so we can update the GDB installation on our host PC without needing to install a new version of gdbserver on the target.

Because most distributions do not package GDB with MIPS architecture support, we need to compile GDB from source. This is accomplished easily using the instructions in the source tarball, which can be downloaded from the GDB Web site. If you get stuck with cross compilation or with the GDB/gdbserver configuration, plenty of good references exist on-line that will help; the Resources section for this article lists a few.

Initial Debugging

Now that we have GDB cross-compiled and installed, let's take a look at debugging the deadlock on the embedded target.

First, run gdbserver on the target and attach to the

deadlocked process:

```
gdbserver :5555 --attach <pid of process>
```

Now, fire up GDB on the host PC:

```
mipsel-linux-uclibc-gdb
```

Once GDB is running, point it at the target's root filesystem and at the file to debug:

```
(gdb) set solib-absolute-prefix /export/shared/rootfs
(gdb) file hello_world
(gdb)
```

Finally, tell GDB to attach to the process running on the target via gdbserver:

```
(gdb) target remote 10.0.0.6:5555
(gdb)
```

If all goes well, now you should be able to explore the running process a little to see what is going on. Given that the process has deadlocked, examining the state of the threads in the process is a good first port of call:

```
(gdb) info threads
Id  Target Id   Frame
33  Thread 737  0x2aac1068 in __lll_lock_wait from libpthread.so.0
32  Thread 738  0x2aac1068 in __lll_lock_wait from libpthread.so.0
31  Thread 739  0x2aac1068 in __lll_lock_wait from libpthread.so.0
....
3   Thread 767  0x2aac1068 in __lll_lock_wait from libpthread.so.0
2   Thread 768  0x2aac1068 in __lll_lock_wait from libpthread.so.0
1   Thread 736  0x2aab953c in pthread_join from libpthread.so.0
(gdb)
```

The omitted threads in the GDB output are all similarly blocking in __lll_lock_wait(), somewhere in the depths of libpthread.so. Clearly, some of these threads must be waiting for a mutex that another thread has not given up—but which threads, and which mutex?

Some examination of the libpthread source in the uClibc tree shows us that __lll_lock_wait() is a low-level wrapper around the Linux futex system call. The prototype for this function is:

```
void __lll_lock_wait (int *futex, int private);
```

On MIPS, the a0 register typically is used for the first argument to a function. So if we examine a0 for each thread that is blocked in __lll_lock_wait(), we should get a good idea of which threads are waiting on which mutexes. That's a good start, but ideally we want to find out which thread currently owns each mutex. How can we manage that?

Going back to the uClibc sources, we can see that __lll_lock_wait() is called from pthread_mutex_lock(). The integer pointer provided to __lll_lock_wait() is actually a pointer to the pthread_mutex_t structure:

```
typedef union
{
  struct __pthread_mutex_s
  {
    int __lock;
    unsigned int __count;
    int __owner;
    int __kind;
    unsigned int __nusers;
    __extension__ union
    {
      int __spins;
      __pthread_slist_t __list;
    };
  } __data;
  char __size[__SIZEOF_PTHREAD_MUTEX_T];
  long int __align;
} pthread_mutex_t;
```

The __owner field looks interesting, and on further investigation it seems that __owner is set to the thread ID (TID) of the thread that is currently holding the mutex.

By combining these two pieces of information (namely the mutex pointer provided to __lll_lock_wait(); and the __owner field two integers on in that structure), we should be able to find out which threads are blocking on which mutexes.

The trouble is that this would be very tedious to iterate

through by hand. Each thread that is blocking in __lll_lock_wait() will need to be selected. Then the contents of register a0 must be queried for the appropriate stack frame of each thread, and the memory at the location pointed to by a0 examined to discover which thread owns the mutex that the thread is waiting for. Even for this trivial program, we have some 32 threads to look at, which is a lot of manual work.

Putting Python into Practice

Rather than driving the debugger by hand, let's instead look at how we can automate the task described above using the GDB Python API. First, we need to be able to iterate over each thread in the process under debugging (the "inferior", in GDB terminology). To do this, we can use the threads() method of the gdb.Inferior class:

```
for process in gdb.inferiors():
    for thread in process.threads():
        print thread
```

That was easy. Now we need to look at the currently executing stack frame for each thread and figure out whether it is waiting on a mutex. We can do this using the gdb module function selected_frame() and the name() method of the gdb.Frame class:

```
for process in gdb.inferiors():
    for thread in process.threads():
        thread.switch()
        frame = gdb.selected_frame()
        if frame.name() == "__lll_lock_wait":
            print "Thread is blocking in __lll_lock_wait"
```

So far, so good. Now that we have a method for programmatically finding each thread that is waiting on a mutex, we need to examine the contents of the a0 register for each of those threads. This should extract a pointer to the mutex structure that the thread is waiting on. Happily, GDB provides a convenience variable, $a0, which we can use to access the a0 register. The gdb module function parse_and_eval() provides API access to convenience variables, amongst other things:

```
for process in gdb.inferiors():
    for thread in process.threads():
        thread.switch()
        frame = gdb.selected_frame()
        if frame.name() == "__lll_lock_wait":
            print "Thread is blocking in __lll_lock_wait"
            a0 = gdb.parse_and_eval("$a0")
```

The last piece of information we need to extract from GDB is the contents of memory at the pointer in the a0 register so that we can determine the __owner field for each mutex in play. Although it's probably cheating to do so, we can fall back on the gdb module function execute() to pass the x command to the GDB command-line interface. This will print the contents of memory to a string that we can parse to find the required information:

```
for process in gdb.inferiors():
    for thread in process.threads():
        thread.switch()
        frame = gdb.selected_frame()
        if frame.name() == "__lll_lock_wait":
            print "Thread is blocking in __lll_lock_wait"
            a0 = gdb.parse_and_eval("$a0")
            s = gdb.execute("x/4d $a0", to_string=True).split()
            s.reverse()
            owner = int(s[1])
```

It's not particularly pretty to look at, but it works. This code splits the string returned from the x command into a whitespace-delimited list. Because GDB may alter the labels used at the start of the output depending on what symbolic information it can extract from the application binary, we then

Listing 2. Python Code for GDB Mutex Debugging

```
from collections import defaultdict

# A dictionary of mutex:owner
mutexOwners = {}

# A dictionary of blocking mutex:(threads..)
threadBlockers = defaultdict(list)

# Print the threads
print "Process threads : "
gdb.execute("info threads")

print "Analysing mutexes..."
# Step through processes running under gdb
for process in gdb.inferiors():

    # Step through each thread in the process
    for thread in process.threads():

        # Examine the thread -- is it blocking on a mutex?
        thread.switch()
        frame = gdb.selected_frame()
        if frame.name() == "__lll_lock_wait":

            # a0 is the first argument passed to the function
            a0 = gdb.parse_and_eval("$a0")
            mutex = int(a0)

            # Make a note of which thread blocks on which mutex
            threadBlockers[mutex].append(thread)

            # Make a note of which thread owns this mutex
            if not mutex in mutexOwners:
                s = gdb.execute("x/4d $a0", to_string=True).split()
                s.reverse()
                mutexOwners[mutex] = int(s[1])

    # Print the results of the analysis
    for mutex in mutexOwners:
        print "  Mutex 0x%x :" % mutex
        print "      -> held by thread : %d" % mutexOwners[mutex]
        s = ["%d" % t.ptid[2] for t in threadBlockers[mutex]]
        print "      -> blocks threads : %s" % ' '.join(s)
```

reverse the list and pull out the second-to-last value. This yields the third integer value in the structure, which in this case is the __owner field of pthread_mutex_t.

All that remains to do now is to plug all of these pieces of data together to provide some useful information. Listing 2 shows the full Python code to do this. Putting it all together:

```
(gdb) source mutex_check.py
Process threads :
Id  Target Id    Frame
33  Thread 737   0x2aac1068 in __lll_lock_wait from libpthread.so.0
32  Thread 738   0x2aac1068 in __lll_lock_wait from libpthread.so.0
....
3   Thread 767   0x2aac1068 in __lll_lock_wait from libpthread.so.0
2   Thread 768   0x2aac1068 in __lll_lock_wait from libpthread.so.0
1   Thread 736   0x2aab953c in pthread_join from libpthread.so.0
Analysing mutexes...
  Mutex 0x401cf0 :
     -> held by thread : 740
     -> blocks threads : 737 738 739 741 742 743 744 745 746 747
                         748 749 750 751 752 753 754 755 756 757
                         758 759 760 761 762 763 764 765 766 767
                         768
  Mutex 0x401d08 :
     -> held by thread : 768
     -> blocks threads : 740
(gdb)
```

The deadlock now becomes very clear. Thread 740 is waiting for a mutex currently owned by thread 768, and thread 768 in turn is waiting for the mutex that thread 740 currently owns. Neither thread can run until the mutex owned by the other becomes available. Returning to the GDB prompt, we can generate backtraces for both threads to gain more insight:

```
(gdb) t 30
[Switching to thread 30 (Thread 740)]
#0  0x2aac1068 in __lll_lock_wait ()
(gdb) bt
#0  0x2aac1068 in __lll_lock_wait ()
#1  0x2aaba568 in pthread_mutex_lock ()
#2  0x00400970 in good_printer (data=0x0) at hello_world.c:45
#3  0x2aab7f9c in start_thread ()
#4  0x2aac2200 in __thread_start ()
Backtrace stopped: frame did not save the PC
(gdb) t 2
[Switching to thread 2 (Thread 768)]
#0  0x2aac1068 in __lll_lock_wait ()
(gdb) bt
#0  0x2aac1068 in __lll_lock_wait ()
#1  0x2aaba568 in pthread_mutex_lock ()
#2  0x00400a04 in bad_printer (data=0x0) at hello_world.c:60
#3  0x2aab7f9c in start_thread ()
#4  0x2aac2200 in __thread_start ()
Backtrace stopped: frame did not save the PC
(gdb)
```

As the backtraces show, the two threads have followed two different code paths to end up in the deadlock situation. Reviewing the code for hello_world in light of this information should allow us to find the bug: bad_printer() is taking the print and statistics locks in the wrong order.

MIPS Registers

The MIPS architecture has 32 general-purpose integer registers. Of these, the hardware architecture specifies that registers 0 and 31 are used for the value zero and the function return address, respectively. The usage of the rest of the registers is entirely defined by the software toolchain.

By convention, however, the use of the general-purpose MIPS registers is quite firmly set to allow software interoperability. For example, registers 4 to 7 are used to pass the first four non-floating-point arguments to functions and are given the names a0 to a3.

Conclusion

Adding a Python API to GDB provides another capable weapon in the Linux debugging arsenal. For embedded systems, where other debugging tools may not be so widely available, a powerful programmatic interface to GDB can make the difference between hours of painstaking debugging and minutes of enjoyable scripting.

Astute readers will have noted that the bug we have discovered in this article is not the only bug in hello_world.c. The task of finding and fixing the remaining bugs is left as an exercise for readers to tackle with their new-found GDB Python knowledge. Have fun!■

Tom Parkin (tom.parkin@gmail.com) has been working with Linux and embedded systems for ten years and is still finding new things to get excited about. When not in front of a computer, he enjoys 10k runs and Real Ale, although not in combination.

Resources

GDB: **www.gnu.org/software/gdb**

GDB/Python Wiki: **sourceware.org/gdb/wiki/PythonGdb**

Tom Tromey's Excellent Blog Posts about GDB and Python: **tromey.com/blog/?cat=17**

OpenWrt's GDB Cross-Compilation Makefile: **https://dev.openwrt.org/browser/trunk/toolchain/gdb/Makefile**

A How-To for GDB/gdbserver Usage: **www.linux.com/archive/feature/121735**

uClibc Project: **uclibc.org**

Linux Futex Information: **kernel.org/doc/man-pages/online/pages/man2/futex.2.html**