eCrash: Debugging without Core Dumps

David Frascone

Abstract

How to use backtrace and a custom library to debug your embedded applications.

Embedded Linux does a good job of bridging the gap between embedded programming and high-level UNIX programming. It has a full TCP stack, good debugging tools and great library support. This makes for a feature-rich development environment. There is one downside, however. How can you debug problems that occur out in the field?

On full-featured operating systems, it's easy to use a core dump to debug a problem that occurs in the field.

On non-embedded UNIX systems, when a program encounters an exception, it outputs all of its current state to a file on the filesystem. This file is usually called core. This core file contains all the memory the program was using at the time the failure occurred. This allows for post-mortem investigation to diagnose the exception.

Typically, on embedded Linux systems, there is no (or very little) persistent disk storage. On all of the systems on which I have worked, there is more RAM than persistent storage. So, getting a core dump is impossible. This article describes some alternatives to core dumps that will allow you to perform post-mortem debugging.

Programs can fail for reasons other than exceptions. Programs can deadlock, or they can have run-away threads that use up all system resources (memory, CPU or other fixed resources). It also would be beneficial to generate some kind of persistent crash file under these situations.

Requirements

So, first we need to come up with the information we want to save. Because of memory constraints, saving all of the process' memory is not an option. If it were, you simply could use core dumps! But, there is other very useful information we can save. At the top of the list is the backtrace of the failed thread.

A backtrace is a list of the functions that were called to get to the current position in the program. Even with the absence of system memory and data, a backtrace can shed light onto what was happening at the time of failure.

Many embedded systems also have logs: lists of errors, warnings and metrics to let you know what happened. Having a post-mortem dump of the last few logs before failure is an invaluable asset in finding the root cause of a failure.

In complex, multithreaded systems, you usually have many mutexes. It could be useful, in the case of a deadlock, to show the state of all the processes' mutexes and semaphores.

Showing memory usage statistics also could help diagnose the problem.

Once we have determined the information we want to save, we still need to come up with where to save it. This will vary greatly from system to system. If your system has no persistent storage at all, perhaps you can output the crash information to a serial terminal or display it on an LCD readout. (We have serious space constraints there!) If your system has CompactFlash, you can save it to a filesystem. Or, if it has raw Flash (an MTD device), you can either save it to a jffs2 filesystem, or maybe to a raw sector or two.

If the crash was not too severe, perhaps the crash could be uploaded to a tftp server or sent to a remote syslog facility.

Now that we have a firm grasp on what we want to save, and locations to which we can save it, let's talk about how we are going to do it!

The Backtrace

In general, getting a backtrace is not as simple as it sounds. Accessing system registers (like the stack pointer) varies from architecture to architecture. Thankfully, the FSF comes to our rescue in GNU's C Standard Library (see the on-line Resources). Libc has three functions that will aid us in retrieving backtraces: backtrace(), backtrace_symbols() and backtrace_symbols_fd().

The backtrace() function populates an array of pointers with a backtrace of the current thread. This, in general, is enough information for debugging, but it is not very pretty.

The backtrace_symbols() function takes the information populated by backtrace() and returns symbolic names (function names). The only problem with backtrace_symbols is that it is not async-signal safe. backtrace_symbols() uses malloc(). Because malloc() uses spinlocks, it is not safe to be called from a signal handler (it could cause a deadlock).

The backtrace_symbols_fd() function attempts to solve the signal issues associated with malloc and output the symbolic information directly to a file descriptor.

Working inside of a Signal Handler

Some functions inside of libc rely on signals themselves: some IO operations, memory allocation and so on. So, we are very limited in what we should do inside of a handler. In our case, we can cheat a little. Because our program already is crashing, a deadlock is not *that* big of a concern. The code in my examples makes use of several not-allowed functions, such as fwrite(), printf() and sprintf(). But, we can work to avoid some of the functions that are prone to deadlock, such as malloc() and backtrace_symbols().

In my opinion, the biggest loss we have is the loss of backtrace_symbols. But, here is where things get easier. You always can implement your own symbol table and look up the functions from the pointers themselves.

In my examples, I sometimes use backtrace_symbols(). I have not seen a deadlock yet, but it *is* possible.

A Simple Backtrace Handler

So, what does the crash handler look like? To get a backtrace, the first thing we need to do is grab our signals. Some of the common ones are SIGSEGV, SIGILL and SIGBUS. Additionally, abort() is usually called in the case of an assertion and generates a SIGABRT.

Then, when a signal occurs, we need to save our backtrace. The following snippet details a simple backtrace function that displays the backtrace to standard output when a crash happens:

```
void signal_handler(int signo)
    {
       void *stack[20];
    int count, i;

    // Shouldn't use printf . . . oh well
    printf("Caught signal %d\n");

    count = backtrace(stack, 20);
        for (i=0; i < count; i++) {
      printf("Frame %2d: %p\n", i+1, stack[i]);
    }
}
int main(...)
{
  ...
  signal(SIGBUS, signal_handler);
  signal(SIGILL, signal_handler);
  signal(SIGSEGV, signal_handler);
  signal(SIGABRT, signal_handler);
}

Caught signal 11
Frame  1:  0x401a84
Frame  2:  0x401d88
...
```

And, here is a similar signal handler, but one that uses backtrace_symbols to print out a prettier backtrace:

```
void signal_handler(int signo)
    {
       void *stack[20];
           char **functions;

    int count, i;

    // Shouldn't use printf . . . oh well
    printf("Caught signal %d\n");

    count = backtrace(stack, 20);
            functions = backtrace_symbols(stack, count);
         for (i=0; i < count; i++) {
      printf("Frame %2d: %s\n", i, functions[i]);
    }
```

```
            free(functions);
}

Caught signal 11
Frame  1:  ./a.out [0x401a84]
Frame  2:  ./a.out [0x401bfa]
...
```

eCrash—A Generic Crash Handler

As I was writing this article, I realized that this was about the fifth time I had written a crash handler. (Why can't all software be open source?) So, I decided to write a quick library to handle crash dumps and provide it for this article. I liked the little library I started with, but I found myself needing more and more features. As I kept extending it, I realized that it was a *very* useful library, that I wanted to be able to leverage on any future project!

I named the new library eCrash and created a SourceForge site for it (see Resources). Since then, I have been extending it, and it now supports dumping multiple threads—using only backtrace, using backtrace and backtrace_symbols, and using backtrace with a user-supplied symbol table to avoid the malloc() inside of backtrace_symbols. The rest of the examples in this article are going to be leveraging eCrash.

eCrash is relatively simple to use. You first call eCrash_Init() from your parent thread. If you have a single-threaded program, you are already finished. A backtrace will be delivered based on your settings in the parameters structure.

If you have a multithreaded program, any thread that wants to be backtraced in a crash (other than the crashing thread) must also call eCrash_RegisterThread(). It is sometimes useful to dump the stacks of all threads when a crash occurs, not only the crashing thread's stack.

With eCrash, you specify where the output should go by setting file descriptors (async safe writes), FILE * streams (not async safe), and/or a filename of the file to output when a crash occurs. eCrash will write to *all* destinations supplied.

eCrash—Gathering Stacks from Other Threads

Obtaining the stack from a thread that did *not* crash is a bit trickier. When a thread registers, it specifies a signal that the thread does not catch or block. eCrash registers a handler for that signal (called the Backtrace Signal).

When eCrash needs to dump a thread (when some other thread has caused an exception), it sends the thread the Backtrace Signal via pthread_kill(). When that signal is caught, the thread saves its backtrace to a global area and continues on. The main exception handler can then read the stack and display it.

What we end up with is a very nice-looking crash dump, showing exactly what was happening in the system when the failure occurred.

eCrash—A Real-World Example

Enough talking—time for some meat. Now, let's take what we have discussed and put it to work. We are going to use the ecrash_test program included in eCrash. That program was designed to break in any one of its threads (generate a segmentation violation by attempting to write to a NULL pointer).

We execute the test program with the following flags:

```
ecrash_test --num_threads=5 --thread_to_crash=3
```

This causes the test program to generate five threads. All but thread number 3 will call a few functions, then go to sleep. Thread 3 will call a few functions (to make the backtrace interesting) and crash.

The crash file generated is shown in Listing 1 and backtrace_symbols() in Listing 2. Due to space constraints, all listings for this article are available on the *Linux Journal* FTP site (ftp://ftp.ssc.com/pub/lj/issue149/8724.tgz).

The crash file has the backtrace of our offending thread (the one that caused the segmentation violation) and the backtraces of all threads on the system.

Now it's time to debug the crash. We will debug this as if the crash happened at a remote site and the system administrator e-mailed you this crash file.

One last thing: in the real world, the executables always are stripped of debugging information. But, that is okay. As long as you keep a copy of the program *with* its debugging information, you can ship a stripped copy of the code, and everything will still work!

So, in the lab, you have your crash file and your program with debugging information. Run **gdb** on the debug version of your program. We know that we have a segmentation violation. So, starting from frame zero of the offending thread, start listing the code as shown Listing 3 (see the *LJ* FTP site).

- Frame 0 is inside of our crash handler—nothing to see here.
- Frame 1 is also inside of the crash handler.
- Frame 2 is still inside of our crash handler.
- Frame 3 shows no source file (it is inside of libc).
- Frame 4 shows the actual crash (inside of crashC).
- Frame 5 shows crashB.
- Frame 6 shows crashA.
- Frame 7 shows ecrash_test_thread.
- And, frames 8 and 9 are where the thread gets created in libc.

As you can see, there is a trick to displaying function pointers with gdb. Simply give it an address and dereference it in a list:

```
(gdb) list *0xWHATEVER
```

This also works with symbolic_names and offsets:

```
(gdb) list *main+100
```

Okay, that was our crashed thread, but what about one of the sleepers? Examine the backtrace from Thread 5, Listing 4 (see the *LJ* FTP site):

- Frame 0 is inside of our backtrace handler.
- Frame 1 still inside of the handler.
- Frame 2 is in libc.
- Frame 3 is in libc.
- Frame 4 is in libc.
- Frame 5 is inside of sleepFuncC—it is showing the for statement as the program counter, because we are outside of the sleep() function. This is notable because the async signal sent to tell the thread to dump its stack caused sleep() to exit prematurely.
- Frame 6 shows sleepFuncB.
- Frame 7 shows sleepFuncA.
- Frame 8 shows crash_test_thread.
- Frame 9 is where the thread gets created in libc (or libpthread).

So, this thread is one of the sleeping threads. Not much to see, but in some cases, this thread's information could be vital to discovering the cause of a crash.

Other Useful Crash Information

Now that we have clobbered to death what a backtrace is, how to produce one, the different methods of displaying one and how to debug a crash with one, it's time to change gears. A crash file can include a lot more information:

- States of mutexes (who is holding the locks—useful for deadlock diagnosis).
- Current error logs.
- Program statistics.
- Memory usage.
- Most recent network packets.

Some of the above items could be useful information for post-mortem debugging. There is one caveat, however. Because we have encountered an exception, something has gone terribly wrong. Our data structures could be corrupt. We could be low on (or out of) memory.

Also, some threads could be deadlocked waiting on mutexes that our crashed thread was holding.

Because some of the data we want to display might generate another exception (if it is corrupted), we want to display the most important information first, then display more and more unsafe information. Also, to prevent information loss, buffers always should be flushed on FILE* streams.

Conclusion

Diagnosing a problem on a deployed embedded system can be a difficult task. But, choosing the right data to save or display in the case of an exception can make the task much easier.

With a relatively small amount of storage, or a remote server, you can save enough post-mortem information to be able to find a failure in your system.

Resources for this article: http://www.linuxjournal.com/article/9139.