

Debugging Linux applications

MOVIAL

Eero Tamminen

Aschwin van der Woude

(copyright Creanor Movial 2002, www.movial.fi)

15th October 2002

Debugging Linux applications

Eero Tamminen

Aschwin van der Woude

(copyright Creanor Movial 2002, www.movial.fi)

15th October 2002

Contents

1	Introduction	2
2	Print statements	2
3	Strace utility	2
4	Ltrace utility	2
5	Using GDB debugger	2
	5.0.1 <i>Preparing for GDB use</i>	2
	5.0.2 <i>Using GDB</i>	3
	5.1 <i>Short introduction to use of gdb</i>	3
	5.2 <i>Gdb problems</i>	4
6	Electric Fence -library	5
7	Memory leakage detection libraries	5
8	Valgrind	5
	8.1 <i>Valgrind problems</i>	6
9	Conclusion	6

1 Introduction

The Linux development environment has several debugging alternatives. This short white-paper explores debugging tools available for debugging applications, ranging from simple print-statement to specialized tools (e.g. memory-debugging).

2 Print statements

Addition of `printf()` statements to your code is a traditional and time-honored way to debug code. Downside is that you will need to modify and recompile the code whenever you want more or less debug information.

3 Strace utility

Strace will output all the kernel calls that the application does and is a great way to find e.g. what file the program is trying to access and whether it succeeded. For instance calls to `read()` and `write()` will show how much data program tried to read/write and how much actually was transferred, it also shows the beginning of the data in question. You can use this without recompilations and it works on any program which you can run.

4 Ltrace utility

Ltrace will output all the *dynamic* library function calls that the application does. It can also show system calls like 'strace'.

5 Using GDB debugger

With *gdb* debugger you can examine all the symbols in the program and program runtime state and follow program function calls. If trace utilities and the source code don't give you enough information to solve the problem, debugger is the next step. *Gdb* is a console tool, but there are some nice debugging GUIs available that work on top of *gdb*. One such is *ddd*.

Gdb can help with debugging programs written in C, C++, Fortran, Java, Chill, assembly and Modula-2. You need to have compiled these programs with the gnu compiler collection (*gcc*) tools.

Besides supporting multiple languages *gdb* also supports multiple hardware architectures, including several embedded hardware architectures. It's also possible to compile a special version of GDB for debugging (Linux) kernel code.

5.0.1 Preparing for GDB use

You need to compile all the C/C++ code you want to debug with debugging information included into the binary (use '-g' and do not use '-fomit-frame-pointer' option when compiling the code) and the code may not be stripped of symbols (do not use '-s' flag in the compilation). Note that all the libraries used by the program should also be compiled this way as missing stack frame pointer can confuse GDB.

It's better if you use static linking (e.g. use '-static' option in your Makefile) because that way gdb won't have trouble finding the symbols for the libraries program uses (for example if you use different libraries with your program than what your compilation machine normally uses).

5.0.2 Using GDB

There are two ways to use a debugger:

- Using debugger to examine the code runtime behavior. You start 'gdb' with the program binary ('gdb <app>') in the program directory. Then you can either run the program inside the gdb with 'run <program arguments>' or attach to an already running instance of the program with 'attach <PID>'. Latter is handy when you don't have working gdb inside the debugging environment. Then you can set breakpoints, examine the code, variables, stack etc and call the program functions.
- Using debugger to examine a process post-mortem 'core' dump. You start 'gdb' with the program binary and core dump ('gdb <app> <core>') in the program code directory. You can examine where the program crashed and what was the state of all the program variables. If your program crashes, but doesn't produce a core file, check 'ulimit -a' to see whether your environment allows core files and fix it with 'ulimit -c unlimited'.
The downside of debugger approach is that debug versions of the binaries are large and statically linked debug ones are huge. If the target doesn't have enough memory for this, you can use 'gdbserver' and stripped binaries on the target. On the host on which you do the debugging, you use gdb with 'target remote' option and give the host gdb the binary with all the debugging symbols.

5.1 Short introduction to use of gdb

You get back to GDB prompt when your program either:

- Crashes.
- Code execution reaches a breakpoint.
- You suspend or otherwise send a signal to the program.

On latter two cases you can use 'cont' to continue the program execution. In the GDB prompt you can do one of the following:

- Set a breakpoint with 'break <function name>'. You can delete breakpoint by saying 'delete <breakpoint number>'.
-□ Examine current program execution trace with 'bt', which will show you where the program execution was interrupted (see above), how it got there and what were the function arguments.
- Move up and down in the execution stack frame by typing 'up' or 'down'.
- Examine program state with either 'info locals' which shows you the state of variables in the current context (function) or use 'print <C expression>' to show you a value of given variable or function. Any valid C expression can be used, even function calls!
- View your program code with 'list <function name>' which will list code of the given function.

You can 'step' through the program code with following commands:

- 'step' will execute the current line and go to next command. If code line is a function call, step will enter the function.
- 'next' works like 'step' but function calls are executed as single instruction.
- 'finish' will execute to the end of current scope (function).
- 'cont' will continue program execution.

Typing return will repeat the previous command. Gdb can also complete function and variable names with TAB key.

5.2 Gdb problems

When optimization is used in code compilation, variable values shown by gdb may sometimes be valid only for variables used on the line that program executed lastly¹, not for the whole scope in where the variables are declared. Use of inlines especially in C++ code (e.g. methods with their body in the header file) may confuse gdb so that it shows either a wrong filename or line for the fault.

In these cases it's better not to optimize the code so much, use only '-O' optimization flag, and forbid inlining completely with the '-fno-default-inline' compilation flag.

¹ Optimization moves variables from memory (stack or heap) into registers and gdb can't follow that as it's not stated in the code and debug symbols.

6 Electric Fence -library

Just by adding '-lefence' to the end of Makefile linker line, your code will be linked with versions of malloc and free functions that will note any overwrites over the allocated area or incorrect frees and which will then immediately SIGSEGV your program, thus getting you into debugger just where the problem is.

Efence will make your program much slower and take enormously more memory (each allocation gets it's own memory page), i.e. you can't run this on the target. Before target testing this is an excellent tool.

7 Memory leakage detection libraries

You can link your code with malloc debugger (e.g. 'dmalloc' or 'mpatrol') library to get information of memory leakage inside the program. These libraries replace the normal allocation functions with their own ones.

Note that Unix/Linux frees all memory allocated by a process when the process exits (unless the memory is shared memory), so one-time allocations are not a problem although these utilities might report those also as 'leakage'. Only leakage that grows within the program lifetime when it should not, is a real problem.

8 Valgrind

Valgrind is a shared library that will run your program code on a simulated x86 CPU. You don't need to link your program to the valgrind library, the valgrind script can do that at runtime to any dynamically linked ' program.

In addition to tools efence and other leak detection libraries (that just replace 'malloc' and 'free' function calls) provide , valgrind x86 simulation can also detect:

- The use of uninitialized memory.
- Reading and writing of freed memory after it has been free'd
- Reading and writing past the end of malloced memory.
- Reading and writing of inappropriate areas on the stack. (Stack-overflows are otherwise very hard to debug. E.g. gdb needs a working stack frame!)
- Memory leaks involving lost pointers to malloced blocks.
- The passing of uninitialized and/or unaddressible memory to system calls.
- The mismatched use of malloc/new/new and free/delete/delete.
- Some possible misuses of the POSIX pthreads API.

Valgrind is the preferred allocation debugger on KDE v3 desktop environment.

² Setting LD_PRELOAD environment variable to point in to *valgrind* library before running the program will make the dynamic linker to load *valgrind* library before your program starts. Dynamic linker will call the dynamic library initialization code which in the case of *valgrind* will move the program execution to x86 CPU simulation within *valgrind*. Thus *valgrind* can observe everything the program does outside kernel space.

8.1 Valgrind problems

Valgrind will slow down the program execution 25-100 times and it will take much more memory. Normally this shouldn't be a problem as you're not going to use this on the target.

Valgrind doesn't work with threaded applications or applications using MMX instructions. It's support for FPU and signals is also sub-optimal.

9 Conclusion

There are many high quality debugging tools available for Linux, each solving a specific area of potential bugs.

Print-statements can be used to pin-point a reproducible bug.
Trace-utilities (strace/ltrace) are used to find the part of the software-stack responsible for the bug, and perhaps the type of bug.

Gdb is a general purpose debugger with many tools to inspect run-time behaviour of applications.

Electric fence finds memory-leaks in an application.
Valgrind finds bugs causing misbehaviour in the use of memory, including buffer- and stack-overflows.

All of these tools are used from the command-line, some of them have graphical frontends for them. These frontends are developed separately from these tools, and thus we left them out of this overview. They do however provide easier access to these tools. Perhaps another paper will go into those frontends.

Creanor Movial
Tietokuja 2, FIN-00330 HELSINKI
tel. +358 9 8567 6400, fax. +358 9 8567 6401

www.movial.fi