Problem-solving best practices for sys admins

Digging Deep

Whether you're the sys admin of a home network or of a company-wide network of dozens or even hundreds of machines, some basic principles of debugging will come in handy. *By Juliet Kemp*

Sometimes you're faced with a brand new system administration problem that you've never met before; sometimes it's an old problem with a new twist. Either way, and regardless of how many machines fall within your sphere of responsibility, just keep a basic set of ideas and processes in mind when you're tracking that mysterious bug through a maze of twisty error messages.

Keeping Track

If you can only do one thing to simplify and speed up problem-solving, keep notes. Clear notes are ideal, but any notes are better than none. Start with a complete bug report of the situation, making sure you're clear on the following issues:

- What commands or situations trigger the problem? You need the exact command or series of commands being used (or a clear and repeatable description of what the user does if it's a GUI).
- What is the expected result?
- What is the actual result?
- Does it happen to many users or just one user?
- Does it happen on different machines or just one machine? (Both this and the previous question are particularly useful in narrowing down problems if you're running any kind of centralized login or shared directory setup.)
- Does the problem happen if you're in a local directory rather than a shared one (if applicable)?
- Can you reproduce it reliably? (If not, keep investigating variables until you can, or you'll never know whether it's really fixed.)
- What has the user tried so far to fix it?

If someone else is reporting the bug, this is a useful set of questions to ask before you start looking for the problem. If it's a bug you've found yourself, it's still a useful checklist to make sure you genuinely understand the problem you're trying to fix and, perhaps most importantly, what it will look like once you *have* fixed it.

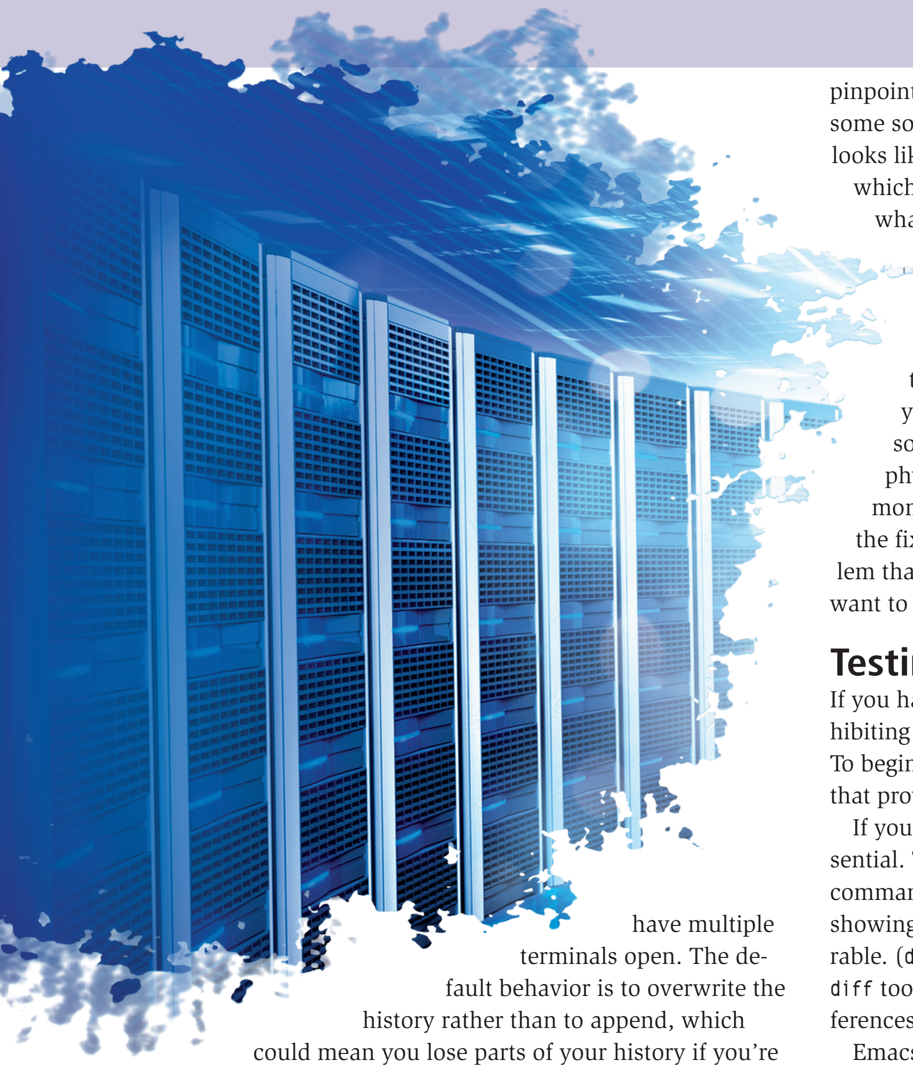
Once you have a clear idea of the problem (see the "Ducks and Bears" box) and have started looking for a solution, make a note every time you make any sort of change. If you make a handful of changes in a hurry, you can check your `~/.bash_history` file (if using Bash; other shells have a similarly named file) to remind yourself what they were. However, the default settings for Bash history are not helpful if you

DUCKS AND BEARS

Rubber Duck debugging or Teddy Bear debugging relies on the fact that simply explaining the problem to someone else is often enough for you to understand it yourself. The other person doesn't actually need to have any idea what you're talking about; a rubber duck or teddy bear works just as well as a real live human – possibly even better because they're better listeners than most humans.

AUTHOR

Juliet Kemp is a systems administrator and writer who has far more experience than she'd like of tracking down bizarre, and occasionally unrepeatable, bugs.



have multiple terminals open. The default behavior is to overwrite the history rather than to append, which could mean you lose parts of your history if you're issuing commands in more than one terminal. If you add the following lines

```
shopt -s histappend
PROMPT_COMMAND='history -n;history -a'
```

to your `.bashrc` file, you'll change the setting to append rather than to overwrite and also set the terminal to write to the history and reload it every time you press Enter (`PROMPT_COMMAND` is executed every time you hit Enter). This means you'll always have access in all terminals to all recently executed commands. Also, you might be able to make similar changes in other shells; check the man page for your shell.

If you're editing a file, it's also a good idea to make a copy of the existing version first. Ideally, you should be using a version control system or some form of centralized setup, like Puppet [1], for your configuration files.

In the heat of bug fixing, you might think you don't need to keep notes as you go because, of course, you'll remember what you did. But, even if you do (in which case, you have a better memory than I), you won't necessarily be able to pinpoint the specific successful change(s) you made without some sort of documentation. Also, you run the risk that what looks like a two-minute fix will turn into a two-hour fix, by which time the probability that you'll remember exactly what you've done has plummeted dramatically.

How you keep your notes, especially in the longer term, is up to you and your personal preferences. A physical notebook has long been a useful tool for any kind of bug fixing and can be helpful for thinking things through and jotting down ideas as they occur to you. A more searchable long-term option is a wiki or some form of electronic note system. If you do use a physical notebook or pieces of paper in the immediate moment, you might want to transfer a concise version of the fix, once you've found it, to a wiki, for example. A problem that's occurred once tends to occur again, and you don't want to have to go through the same process all over again.

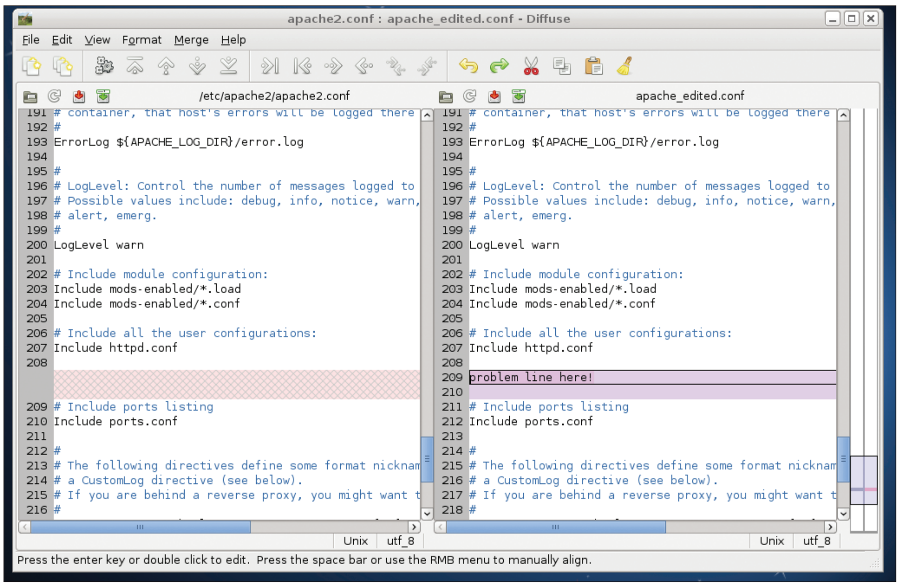
Testing Hints

If you have more than one machine and only one of them is exhibiting the problem, you have a great comparison test setup. To begin, check the configuration of both machines and see if that provides any clues as to what the problem might be.

If you're comparing config files, some form of diff tool is essential. The original, of course is `diff`, but you might prefer its command-line cousin `sdiff` because it's easier to read by showing differences side by side, and it is a bit more configurable. (`diff -y` achieves the same thing). The `vimdiff` or `gvimdiff` tools are also options; both use color to help show any differences.

Emacs also has a diff mode (`M-x ediff-buffers`). If you prefer a GUI, you could try Meld, Guiffy (which also works on Mac and Linux), `diffuse` (Figure 1), TkDiff, or `kdiff3`. Other good first-line tools to try first are `ps`, `top`, and `df` to check for obvious issues and then to compare with other machines. A significant number of problems boil down to disk, memory, or CPU issues.

As with bug-finding in coding, a good way to locate a bug in a system is to create a test for it and then narrow down that


apache2.conf : apache_edited.conf - Diffuse
File
Edit
View
Format
Merge
Help
/etc/apache2/apache2.conf
apache_edited.conf
Unix
utf_8
Unix
utf_8

```
# container, that host's errors will be logged there
#
ErrorLog ${APACHE_LOG_DIR}/error.log

#
# LogLevel: Control the number of messages logged to
# Possible values include: debug, info, notice, warn,
# alert, emerg.
#
LogLevel warn

# Include module configuration:
Include mods-enabled/*.load
Include mods-enabled/*.conf

# Include all the user configurations:
Include httpd.conf


# Include ports listing
Include ports.conf

#
# The following directives define some format nicknam
# a CustomLog directive (see below).
# If you are behind a reverse proxy, you might want t
#
```

```
# container, that host's errors will be logged there
#
ErrorLog ${APACHE_LOG_DIR}/error.log

#
# LogLevel: Control the number of messages logged to
# Possible values include: debug, info, notice, warn,
# alert, emerg.
#
LogLevel warn

# Include module configuration:
Include mods-enabled/*.load
Include mods-enabled/*.conf

# Include all the user configurations:
Include httpd.conf

problem line here!

# Include ports listing
Include ports.conf

#
# The following directives define some format nicknam
# a CustomLog directive (see below).
# If you are behind a reverse proxy, you might want t
#
```

Press the enter key or double click to edit. Press the space bar or use the RMB menu to manually align.

Figure 1: Diffuse showing the differences between two files.

test as much as possible while the system still exhibits the problem.

Here are a few questions to help you narrow down the possibilities:

- Can you test a GUI program from the command line?
- Have you encountered a similar problem before?
- What happens if you shut down everything else running on the same machine?
- What happens if you create a new user with a super-minimalist setup and try to reproduce the bug from there?
- What happens if you clear the user's Bash/tcsh configuration (make sure you don't lose it altogether…)?
- Can you create a stripped-down version of the problem situation?
- What happens if you temporarily turn off iptables or any other firewalling/networking security you have in place for networking components?

If the problem is connected in any way to a service that you're running (e.g., Apache), another useful step when testing is to boost logging output temporarily. Exactly how to do this will vary between different pieces of software, but the main config file will usually have a logging output option, and you can check the appropriate man page or website for information on which settings produce the most useful output (see also "Booting Up Gradually").

Once you have your smallest possible test situation, you can start making changes. To begin, make a single change, then run the test again. If you make multiple changes at once, you'll come to a point at which you don't know what made the crucial difference. Make a note of what you did and what happened, then change something else and test again.

When running tests, be aware of a couple of things: If you've been looking at multiple machines, make sure the change and the test are happening on the same machine. Also, be certain that the change has actually been applied and that the configuration file (or whatever) has been re-read. If in doubt, restart the relevant service or log in again as the user whose configuration you're changing.

Languages You Don't Know

Sometimes you will find yourself trying to debug a program in a language you don't really know. If it's fairly complicated, your might want to recruit someone to help you (or make the fix for you); however, you can try the following things first:

Back up. Make sure you back up the code before you start – if you intend to make any changes at all.

Google the error messages. If you're lucky, someone has already experienced this problem and discovered how to fix it. You might be able to copy the fix straight in without learning anything more about the code or the language.

Is it a compiled or an interpreted language? The distinction is less firm than it once was, but it's still useful to distinguish between a compiled language, such as C or Java, in which code is rendered into a machine-specific format before being executed, and an interpreted language, such as PHP or Perl, in which code is executed as you wrote it and rendered into machine-specific format at execution time. With compiled languages, you'll have a source code file and an object code executable, and to make changes, you need to edit the source code, recompile it, and execute the resulting binary to test it:

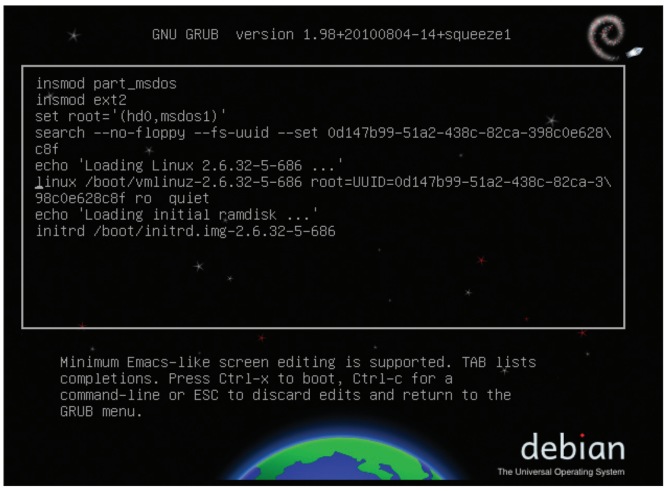
```
> javac helloworld.java
> java helloworld
Hello, World!
```

With interpreted languages, you edit the code then run it directly:

```
> perl helloworld.pl
Hello, World!
```

To debug compiled languages, it's a good idea to keep a copy of both the source code and the object code before you start messing around. Also, recompile the source and test it before you start, just to make sure you really do have the correct source code for your problem object code file.

Use a debugger. If you can use a decent debugger on the code or program in question, this is a great place to start. Many



```
GNU GRUB  version 1.98+20100804-14+squeeze1

insmod part_msdos
insmod ext2
set root='(hd0,msdos1)'
search --no-floppy --fs-uuid --set 0d147b99-51a2-438c-82ca-398c0e628\
c8f
echo 'Loading Linux 2.6.32-5-686 ...'
linux /boot/vmlinuz-2.6.32-5-686 root=UUID=0d147b99-51a2-438c-82ca-3\
98c0e628c8f ro  quiet
echo 'Loading initial ramdisk ...'
initrd /boot/initrd.img-2.6.32-5-686

    Minimum Emacs-like screen editing is supported. TAB lists
    completions. Press Ctrl-x to boot, Ctrl-c for a
    command-line or ESC to discard edits and return to the
    GRUB menu.
```

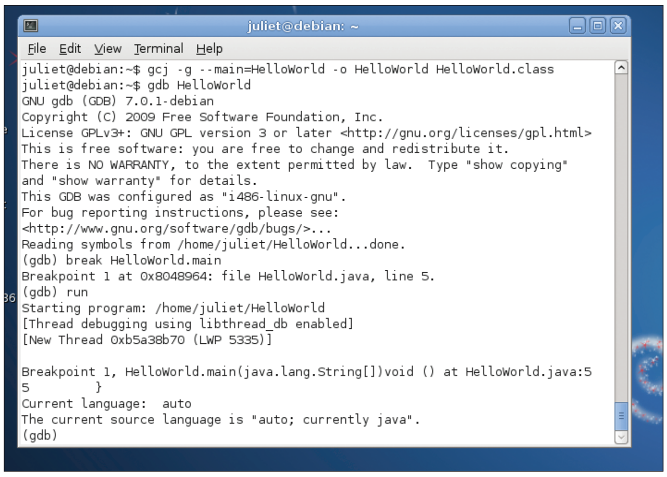
Figure 2: Editing the bootup line on Debian Squeeze.

BOOTING UP GRADUALLY

Boot problems, or problems with services that start automatically, can be particularly difficult to deal with because the problem often occurs before you can get to it and fix it. (This is particularly true with firewall and security or login problems, which can result in being locked out of the machine altogether.)

To get at these problems, try booting into `/bin/sh` rather than `init`, which on most Linux distros is still the first task run by the kernel to start up all your normal services. This way, you get a basic shell prompt at a very early stage of boot. With GRUB, do this by hitting *e* at the GRUB boot screen, then edit the boot line that begins with *kernel* or *linux* (Figure 2). Add `init=/bin/sh` to the end of that line and boot the system (on older versions of GRUB, hit Return then *b*; on newer versions, press Ctrl+X). Be aware that if you do this, you'll have very few services up and running, which is the point of the exercise. For debugging purposes, you can start things up one by one, maximizing your chances of locating the error as quickly as possible.

In this case, you definitely want to keep careful notes of anything you change, and change it back again if it doesn't do the trick *before* you reboot. Otherwise, you could simply create further problems for yourself.



juliet@debian: ~

File Edit View Terminal Help

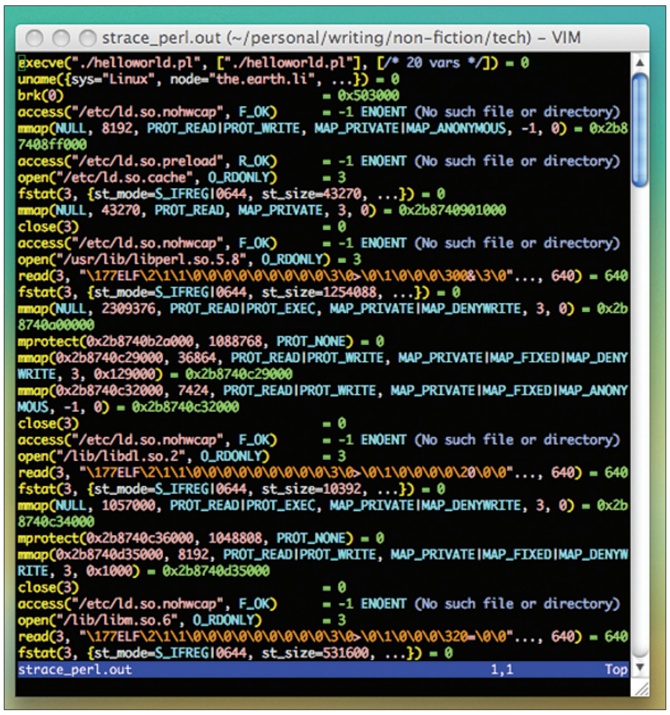
```
juliet@debian:~$ gcj -g --main=HelloWorld -o HelloWorld HelloWorld.class
juliet@debian:~$ gdb HelloWorld
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/juliet/HelloWorld...done.
(gdb) break HelloWorld.main
Breakpoint 1 at 0x8048964: file HelloWorld.java, line 5.
(gdb) run
Starting program: /home/juliet/HelloWorld
[Thread debugging using libthread_db enabled]
[New Thread 0xb5a38b70 (LWP 5335)]

Breakpoint 1, HelloWorld.main(java.lang.String[])void () at HelloWorld.java:5
5         }
Current language:  auto
The current source language is "auto; currently java".
(gdb)
```

Figure 3: gdb in action.

languages have a standard debugger available; if you don't have one that's language specific, try `gdb`, which although a bit archaic in some ways, basically works for several languages, including C, C++, Java (if compiled with `gcj`), and Fortran. The `bt` command is particularly useful if you're using `gdb` because it prints out a backward stack trace, which can help you to figure out where the code is crashing (Figure 3). Setting breakpoints (points where the code stops) with the `break` command can also be useful, but be aware that this is only useful if the code you're looking at is non-iterative.

Add print lines. If you don't have a debugger, you can try editing the code and adding some kind of print output. Check the documentation or Google to find out how to add a print line for a specific language to create your own "breakpoints"; add output lines throughout the code until you narrow down the point at which something is going wrong (between output lines D and E, for example). You can also use output lines to check the values of different variables or data structures.



strace_perl.out (~/personal/writing/non-fiction/tech) – VIM

```
execve("./helloworld.pl", ["./helloworld.pl"], [/* 20 vars */]) = 0
uname({sys="Linux", node="the.earth.li", ...}) = 0
brk(0)                                  = 0x503000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2b8
7408ff000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=43270, ...}) = 0
mmap(NULL, 43270, PROT_READ, MAP_PRIVATE, 3, 0) = 0x2b8740901000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/usr/lib/libperl.so.5.8", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300&\3\0"..., 640) = 640
fstat(3, {st_mode=S_IFREG|0644, st_size=1254088, ...}) = 0
mmap(NULL, 2309376, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x2b
8740a00000
mprotect(0x2b8740b2a000, 1088768, PROT_NONE) = 0
mmap(0x2b8740c29000, 36864, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENY
WRITE, 3, 0x129000) = 0x2b8740c29000
mmap(0x2b8740c32000, 7424, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONY
MOUS, -1, 0) = 0x2b8740c32000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libdl.so.2", O_RDONLY)       = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\20\0\0"..., 640) = 640
fstat(3, {st_mode=S_IFREG|0644, st_size=10392, ...}) = 0
mmap(NULL, 1057000, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x2b
8740c34000
mprotect(0x2b8740c36000, 1048808, PROT_NONE) = 0
mmap(0x2b8740d35000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYW
RITE, 3, 0x1000) = 0x2b8740d35000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libm.so.6", O_RDONLY)        = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320=\0\0"..., 640) = 640
fstat(3, {st_mode=S_IFREG|0644, st_size=531600, ...}) = 0
strace_perl.out                                              1,1           Top
```

Figure 4: strace output from a Perl program.

Log. Logging is another option to get output from code. Again, search the web for logging options for different languages.

Use the command line. If possible, run the problem program from the command line and send output to the terminal.

Break down the source. Beyond breakpoints and printing debug lines, another testing option that narrows down the problem (especially if you can't edit the code directly) is to copy the source and cut it down as far as possible until you have the problem and nothing else. In a large program, this might not be possible.

Proceed carefully. Once you think you've tracked down the problem, make a single change at a time before testing again.

Use tracing tools. Strace (which tracks system calls) can help determine whether system calls are the source of the problem.

```
strace -o output.txt program
```

will get strace output to the file `output.txt`. Some editors (including Vim) will do syntax highlighting for strace output, which is helpful (Figure 4). Similarly, `ltrace` provides information on library calls being made, these are both particularly helpful if the problem has something to do with the interaction of your code with the larger system. Even in other situations it could give you useful context (e.g., if the program fails when reading in a particular file, this suggests that the problem is either with the file or with that section of code).

Get help. If you're still stuck, talk to people who already know the language. Some problems are easy to track down with a little general knowledge and careful testing; some really do need more specialised knowledge. If you're asking for help, make sure you've described the problem clearly; give the sort of bug report you'd like to receive.

Avoiding Problems

The ideal solution, of course, is to avoid problems arising in the first place. Although this is never going to be entirely possible, you can certainly do your best to minimize them.

A good testing regime whenever you're making changes is important. Test regularly as you install or change something, and test on multiple machines, if possible. Good note-taking so you'll have something to jog your memory will help if you run into a problem further down the line. Always write documentation of any sort as if you're writing for someone else because in six months' time, you'll *be* someone else: someone who doesn't remember all that well what they did six months ago.

Keep config files in version control, or (if appropriate for your setup) a centralized config management system such as Puppet. This makes keeping track of your changes hugely easier, as long as you remember to use it every time!

If All Else Fails

Take a break. Sometimes all it takes to find the solution to a problem is to walk away from it for a moment: Take the time to have a drink of water, a cup of coffee, or a snack. ■■■

INFO

[1] Puppet: *http://projects.puppetlabs.com/*