CHAPTER 1

Best Practices and Initial Investigation

1.1 Introduction

Your boss is screaming, your customers are screaming, you're screaming … Whatever the situation, there is a problem, and you need to solve it. Remember those old classic MUD games? For those who don't, a Multi-User Dungeon or MUD was the earliest incarnation of the online video game. Users played the game through a completely non-graphical text interface that described the surroundings and options available to the player and then prompted the user with what to do next.

```
You are alone in a dark cubicle. To the North is your boss's office, to
the West is your Team Lead's cubicle, to the East is a window opening
out to a five-floor drop, and to the South is a kitchenette containing
a freshly brewed pot of coffee. You stare at your computer screen in
bewilderment as the phone rings for the fifth time in as many minutes
indicating that your users are unable to connect to their server.

Command>
```

What will you do? Will you run toward the East and dive through the open window? Will you go grab a hot cup of coffee to ensure you stay alert for the long night ahead? A common thing to do in these MUD games was to examine your surroundings further, usually done by the `look` command.

```
Command> look

Your cubicle is a mess of papers and old coffee cups. The message
waiting light on your phone is burnt out from flashing for so many
months. Your email inbox is overflowing with unanswered emails. On top
of the mess is the brand new book you ordered entitled "Self-Service
Linux." You need a shower.

Command> read book "Self-Service Linux"

You still need a shower.
```

This tongue-in-cheek MUD analogy aside, what can this book really do for you? This book includes chapters that are loaded with useful information to help you diagnose problems quickly and effectively. This first chapter covers best practices for problem determination and points to the more in-depth information found in the chapters throughout this book. The first step is to ensure that your Linux system(s) are configured for effective problem determination.

1.2 Getting Your System(s) Ready for Effective Problem Determination

The Linux problem determination tools and facilities are free, which begs the question: Why not install them? Without these tools, a simple problem can turn into a long and painful ordeal that can affect a business and/or your personal time. Before reading through the rest of the book, take some time to make sure the following tools are installed on your system(s). These tools are just waiting to make your life easier and/or your business more productive:

☞ **strace:** The strace tool traces the system calls, special functions that interact with the operating system. You can use this for many types of problems, especially those that relate to the operating system.

☞ **ltrace:** The ltrace tool traces the functions that a process calls. This is similar to strace, but the called functions provide more detail.

☞ **lsof:** The lsof tool lists all of the open files on the operating system (OS). When a file is open, the OS returns a numeric file descriptor to the process to use. This tool lists all of the open files on the OS with their respective process IDs and file descriptors.

☞ **top:** This tool lists the "top" processes that are running on the system. By default it sorts by the amount of current CPU being consumed by a process.

☞ **traceroute/tcptraceroute:** These tools can be used to trace a network route (or at least one direction of it).

☞ **ping:** Ping simply checks whether a remote system can respond. Sometimes firewalls block the network packets ping uses, but it is still very useful.

- ☞ **hexdump or equivalent:** This is simply a tool that can display the raw contents of a file.
- ☞ **tcpdump and/or ethereal:** Used for network problems, these tools can display the packets of network traffic.
- ☞ **GDB:** This is a powerful debugger that can be used to investigate some of the more difficult problems.
- ☞ **readelf:** This tool can read and display information about various sections of an Executable and Linking Format (ELF) file.

These tools (and many more) are listed in Appendix A, "The Toolbox," along with information on where to find these tools. The rest of this book assumes that your systems have these basic Linux problem determination tools installed. These tools and facilities are free, and they won't do much good sitting quietly on an installation CD (or on the Internet somewhere). In fact, this book will self-destruct in five minutes if these tools are not installed.

Now of course, just because you have a tool in your toolbox, it doesn't mean you know how to use it in a particular situation. Imagine a toolbox with lots of very high quality tools sitting on your desk. Suddenly your boss walks into your office and asks you to fix a car engine or TV. You know you have the tools. You might even know what the tools are used for (that is, a wrench is used for loosening and tightening bolts), but could you fix that car engine? A toolbox is not a substitute for a good understanding of how and when to use the tools. Understanding how and when to use these tools is the main focus of this book.

1.3 The Four Phases of Investigation

Good investigation practices should balance the need to solve problems quickly, the need to build your skills, and the effective use of subject matter experts. The need to solve a problem quickly is obvious, but building your skills is important as well.

Imagine walking into a library looking for information about a type of hardwood called "red oak." To your surprise, you find a person who knows absolutely everything about wood. You have a choice to make. You can ask this person for the information you need, or you can read through several books and resources trying to find the information on your own. In the first case, you will get the answer you need right away...you just need to ask. In the second case, you will likely end up reading a lot of information about hardwood on your quest to find information about red oak. You're going to learn more about hardwood, probably the various types, relative hardness, and what each is used for. You might even get curious and spend time reading up on the other types of hardwood. This peripheral information can be very helpful in the future, especially if you often work with hardwood.

The next time you need information about hardwood, you go to the library again. You can ask the mysterious and knowledgeable person for the answer or spend some time and dig through books on your own. After a few trips to the library doing the investigation on your own, you will have learned a lot about hardwood and might not need to visit the library any more to get the answers you need. You've become an expert in hardwood. Of course, you'll use your new knowledge and power for something nobler than creating difficult decisions for those walking into a library.

Likewise, every time you encounter a problem, you have a choice to make. You can immediately try to find the answer by searching the Internet or by asking an expert, or you can investigate the problem on your own. If you investigate a problem on your own, you will increase your skills from the experience regardless of whether you successfully solve the problem.

Of course, you need to make sure the skills that you would learn by finding the answer on your own will help you again in the future. For example, a physician may have little use for vast knowledge of hardwood ... although she or he may still find it interesting. For a physician that has one question about hardwood every 10 years, it may be better to just ask the expert or look for a shortcut to get the information she or he needs.

The first section of this chapter will outline a useful balance that will solve problems quickly and in many cases even faster than getting a subject matter expert involved (from here on referred to as an *expert*). How is this possible? Well, getting an expert usually takes time. Most experts are busy with numerous other projects and are rarely available on a minute's notice. So why turn to them at the first sign of trouble? Not only can you investigate and resolve some problems faster on your own, you can become one of the experts of tomorrow.

There are four phases of problem investigation that, when combined, will both build your skills and solve problems quickly and effectively.

1. Initial investigation using your own skills.
2. Search for answers using the Internet or other resource.
3. Begin deeper investigation.
4. Ask a subject matter expert for help.

The first phase is an attempt to diagnose the problem on your own. This ensures that you build some skill for every problem you encounter. If the first attempt takes too long (that is, the problem is urgent and you need an immediate solution), move on to the next phase, which is searching for the answer using the Internet. If that doesn't reveal a solution to the problem, don't get an expert involved just yet. The third phase is to dive in deeper on your own. It will help to build some deep skill, and your homework will also be appreciated by an expert should you need to get one involved. Lastly, when the need arises, engage an expert to help solve the problem.

The urgency of a problem should help to guide how quickly you go through the phases. For example, if you're supporting the New York Stock Exchange and you are trying to solve a problem that would bring it back online during the peak hours of trading, you wouldn't spend 20 minutes surfing the Internet looking for answers. You would get an expert involved immediately.

The type of problem that occurred should also help guide how quickly you go through the phases. If you are a casual at-home Linux user, you might not benefit from a deep understanding of how Linux device drivers work, and it might not make sense to try and investigate such a complex problem on your own. It makes more sense to build deeper skills in a problem area when the type of problem aligns with your job responsibilities or personal interests.

1.3.1 Phase #1: Initial Investigation Using Your Own Skills

Basic information you should always make note of when you encounter a problem is:

☞ The exact time the problem occurred

☞ Dynamic operating system information (information that can change frequently over time)

The exact time is important because some problems are related to an event that occurred at that time. A common example is an errant cron job that randomly kills off processes on the system. A cron job is a script or program that is run by the cron daemon. The cron daemon is a process that runs in the background on Linux and Unix systems and runs programs or scripts at specific and configurable times (refer to the Linux man pages for more information about cron). A system administrator can accidentally create a cron job that will kill off processes with specific names or for a certain set of user IDs. As a non-privileged user (a user without super user privileges), your tool or application would simply be killed off without a trace. If it happens again, you will want to know what time it occurred and if it occurred at the same time of day (or week, hour, and so on).

The exact time is also important because it may be the only correlation between the problem and the system conditions at the time when the problem occurred. For example, an application often crashes or produces an error message when it is affected by low virtual memory. The symptom of an application crashing or producing an error message can seem, at first, to be completely unrelated to the current system conditions.

The dynamic OS information includes anything that can change over time without human intervention. This includes the amount of free memory, the amount of free disk space, the CPU workload, and so on. This information is important enough that you may even want to collect it any time a serious problem occurs. For example, if you don't collect the amount of free virtual memory when a problem occurs, you might never get another chance. A few minutes or hours later, the system resources might go back to normal, eliminating any evidence that the system was ever low on memory. In fact, this is so important that distributions such as SUSE LINUX Enterprise Server continuously run sar (a tool that displays dynamic OS information) to monitor the system resources. Sar is a special tool that can collect, report, or save information about the system activity.

The dynamic OS information is also a good place to start investigating many types of problems, which are frequently caused by a lack of resources or changes to the operating system. As part of this initial investigation, you should also make a note of the following:

☞ **What you were doing when the problem occurred.** Were you installing software? Were you trying to start a Web server?

☞ **A problem description.** This should include a description of what happened and a description of what was supposed to happen. In other words, how do you know there was a problem?

☞ **Anything that may have triggered the problem**. This will be pretty problem-specific, but it's worthwhile to think about it when the problem is still fresh in your mind.

☞ **Any evidence that may be relevant.** This includes error logs from an application that you were using, the system log (/var/log/messages), an error message that was printed to the screen, and so on. You will want to protect any evidence (that is, make sure the relevant files don't get deleted until you solve the problem).

If the problem isn't too serious, then just make a mental note of this information and continue the investigation. If the problem is very serious (has a major impact to a business), write this stuff down or put it into an investigation log (an investigation log is covered in detail later in this chapter).

If you can reproduce the problem at will, strace and ltrace may be good tools to start with. The strace and ltrace utilities can trace an application from the command line, or they can trace a running process. The `strace` command traces all of the system calls (special functions that interact with the operating system), and `ltrace` traces functions that a program called. The strace tool is probably the most useful problem investigation tool on Linux and is covered in more detail in Chapter 2, "strace and System Call Tracing Explained."

Every now and then you'll run into a problem that occurs once every few weeks or months. These problems usually occur on busy, complex systems, and even though they are rare, they can still have a major impact to a business and your personal time. If the problem is serious and cannot be reproduced, be sure to capture as much information as possible given that it might be your only chance. Also if the problem can't be reproduced, you should start writing things down because you might need to refer to the information weeks or months into the future. For these types of problems, it may be worthwhile to collect a lot of information about the OS (including the software versions that are installed on it) considering that the problem could be related to something else that may change over weeks or months of time. Problems that take weeks or months to resolve can span several major changes or upgrades to the system, making it important to keep track of the original conditions under which the problem occurred.

Collecting the right OS information can involve running many OS commands, too many for someone to run when the need arises. For your convenience, this book comes with a data collection script that can gather an enormous amount of information about the operating system in a very short period of time. It will save you from having to remember each command and from having to type each command in to collect the right information.

The data collection script is particularly useful in two situations. The first situation is that you are investigating a problem on a remote customer system that you can't log in to. The second situation is a serious problem on a local system that is critical to resolve. In both cases, the script is useful because it will usually gather all the OS information you need to investigate the problem with a single run.

When servicing a remote customer, it will reduce the number of initial requests for information. Without a data collection script, getting the right information for a remote problem can take many emails or phone calls. Each time you ask for more information, the information that is collected is older, further from the time that the problem occurred.

The script is easy to modify, meaning that you can add commands to collect information about specific products (including yours if you have any) or applications that may be important. For a business, this script can improve the efficiency of your support organization and increase the level of customer satisfaction with your support.

Readers that are only using Linux at home may still find the script useful if they ever need to ask for help from a Linux expert. However, the script is certainly aimed more at the business Linux user. For this reason, there is more information on the data collection script in Appendix B, "Data Collection Script" (for the readers who support or use Linux in a business setting).

Do not underestimate the importance of doing an initial investigation on your own, even if the information you need to solve the problem is on the Internet. You will learn more investigating a problem on your own, and that earned knowledge and experience will be helpful for solving problems again in the future. That said, make sure the information you learn is in an area that you will find useful again. For example, improving your skills with strace is a very worthwhile exercise, but learning about a rare problem in a device driver is probably not worth it for the average Linux user. An initial investigation will also help you to better understand the problem, which can be helpful when trying to find the right information on the Internet. Of course, if the problem is urgent, use the appropriate resources to find the right solution as soon as possible.

1.3.1.1 Did Anything Change Recently?

Everything is working as expected and then suddenly, a problem occurs. The first question that people usually ask is "Did anything change recently?" The fact of the matter is that something either changed or something triggered the problem. If something changed and you can figure out what it was, you might have solved the problem and avoided a lengthy investigation.

In general, it is very important to keep changes to a production environment to a minimum. When changes are necessary, be sure to notify the system users of any changes in advance so that any resulting impact will be easier for them to diagnose. Likewise, if you are a user of a system, look to your system administrator to give you a heads up when changes are made to the system. Here are some examples of changes that can cause problems:

☞ A recent upgrade or change in the kernel version and/or system libraries and/or software on the system (for example, a software upgrade). The change could introduce a bug or a change in the (expected) behavior of the operating system. Either can affect the software that runs on the system.

- ☞ Changes to kernel parameters or tunable values can cause changes to behavior of the operating system, which can in turn cause problems for software that runs on the system.
- ☞ Hardware changes. Disks can fail causing a major outage or possibly just a slowdown in the case of a RAID. If more memory is added to the system and applications start to fail, it could be the result of bad memory. For example, gcc is one of the tools that tend to crash with bad memory.
- ☞ Changes in workload (that is, more users suddenly going to a particular Web site) may push the system close to the limit of its resources. Increases in workload can consume the last bit of memory, causing problems for any software that could be running on the system.

One of the best ways to detect changes to the system is to periodically run a script or tool that collects important information about the system and the software that runs on it. When a difficult problem occurs, you might want to start with a quick comparison of the changes that were recently made on the system — if nothing else, to rule them out as candidates to investigate further.

Using information about changes to the system requires a bit of work up front. If you don't save historical information about the operating environment, you won't be able to compare it to the current information when something goes wrong. There are some useful tools such as tripwire that can help to keep a history of good, known configuration states.

Another best practice is to track any changes to configuration files in a revision control system such as CVS. This will ensure that you can "go back" to a stable point in the system's past. For example, if the system were running smoothly three weeks ago but is unstable now, it might make sense to go back to the configuration three weeks prior to see if the problems are due to any configuration changes.

1.3.2 Phase #2: Searching the Internet Effectively

There are three good reasons to move to this phase of investigation. The first is that your boss and/or customer needs immediate resolution of a problem. The second reason is that your patience has run out, and the problem is going in a direction that will take a long time to investigate. The third is that the type of problem is such that investigating it on your own is not going to build useful skills for the future.

Using what you've learned about the problem in the first phase of investigation, you can search online for similar problems, preferably finding the identical problem already solved. Most problems can be solved by searching the Internet using an engine such as Google, by reading frequently asked question (FAQ) documents, HOW-TO documents, mailing-list archives, USENET archives, or other forums.

1.3.2.1 Google When searching, pick out unique keywords that describe the problem you're seeing. Your keywords should contain the *application name* or "kernel" + *unique keywords from actual output* + *function name where problem occurs* (if known). For example, keywords consisting of "kernel Oops sock_poll" will yield many results in Google.

There is so much information about Linux on the Internet that search engine giant Google has created a special search specifically for Linux. This is a great starting place to search for the information you want - `http://www.google.com/linux`.

There are also some types of problems that can affect a Linux user but are not specific to Linux. In this case, it might be better to search using the main Google page instead. For example, FreeBSD shares many of the same design issues and makes use of GNU software as well, so there are times when documentation specific to FreeBSD will help with a Linux related problem.

1.3.2.2 USENET USENET is comprised of thousands of *newsgroups* or discussion groups on just about every imaginable topic. USENET has been around since the beginning of the Internet and is one of the original services that molded the Internet into what it is today. There are many ways of reading USENET newsgroups. One of them is by connecting a software program called a *news reader* to a USENET *news server*. More recently, Google provided *Google Groups* for users who prefer to use a Web browser. Google Groups is a searchable archive of most USENET newsgroups dating back to their infancies. The search page is found at `http://groups.google.com` or off of the main page for Google. Google Groups can also be used to post a question to USENET, as can most news readers.

1.3.2.3 Linux Web Resources There are several Web sites that store searchable Linux documentation. One of the more popular and comprehensive documentation sites is The Linux Documentation Project: `http://tldp.org`.

The Linux Documentation Project is run by a group of volunteers who provide many valuable types of information about Linux including FAQs and HOW-TO guides.

There are also many excellent articles on a wide range of topics available on other Web sites as well. Two of the more popular sites for articles are:

☞ Linux Weekly News – `http://lwn.net`
☞ Linux Kernel Newbies – `http://kernelnewbies.org`

The first of these sites has useful Linux articles that can help you get a better understanding of the Linux environment and operating system. The second Web site is for learning more about the Linux kernel, not necessarily for fixing problems.

1.3.2.4 Bugzilla Databases

Inspired and created by the Mozilla project, Bugzilla databases have become the most widely used bug tracking database systems for all kinds of GNU software projects such as the GNU Compiler Collection (GCC). Bugzilla is also used by some distribution companies to track bugs in the various releases of their GNU/Linux products.

Most Bugzilla databases are publicly available and can, at a minimum, be searched through an extensive Web-based query interface. For example, GCC's Bugzilla can be found at `http://gcc.gnu.org/bugzilla`, and a search can be performed without even creating an account. This can be useful if you think you've encountered a real software bug and want to search to see if anyone else has found and reported the problem. If a match is found to your query, you can examine and even track all the progress made on the bug.

If you're sure you've encountered a real software bug, and searching does not indicate that it is a known issue, do not hesitate to open a new bug report in the proper Bugzilla database. Open source software is community-based, and reporting bugs is a large part of what makes the open source movement work. Refer to investigation Phase 4 for more information on opening a bug reports.

1.3.2.5 Mailing Lists

Mailing lists are related closely to USENET newsgroups and in some cases are used to provide a more user friendly front-end to the lesser known and less understood USENET interfaces. The advantage of mailing lists is that interested parties explicitly *subscribe* to specific lists. When a posting is made to a mailing list, everyone subscribed to that list will receive an email. There are usually settings available to the subscriber to minimize the impact on their inboxes such as getting a daily or weekly digest of mailing list posts.

The most popular Linux related mailing list is the *Linux Kernel Mailing List* (lkml). This is where most of the Linux pioneers and gurus such as Linux Torvalds, Alan Cox, and Andrew Morton "hang out." A quick Google search will tell you how you can subscribe to this list, but that would probably be a bad idea due to the high amount of traffic. To avoid the need to subscribe and deal with the high traffic, there are many Web sites that provide fancy interfaces and searchable archives of the lkml. The main one is `http://lkml.org`.

There are also sites that provide summaries of discussions going on in the lkml. A popular one is at Linux Weekly News (`lwn.net`) at `http://lwn.net/Kernel`.

As with USENET, you are free to post questions or messages to mailing lists, though some require you to become a subscriber first.

1.3.3 Phase #3: Begin Deeper Investigation (Good Problem Investigation Practices)

If you get to this phase, you've exhausted your attempt to find the information using the Internet. With any luck you've picked up some good pointers from the Internet that will help you get a jump start on a more thorough investigation.

Because this is turning out to be a difficult problem, it is worth noting that difficult problems need to be treated in a special way. They can take days, weeks, or even months to resolve and tend to require much data and effort. Collecting and tracking certain information now may seem unimportant, but three weeks from now you may look back in despair wishing you had. You might get so deep into the investigation that you forget how you got there. Also if you need to transfer the problem to another person (be it a subject matter expert or a peer), they will need to know what you've done and where you left off.

It usually takes many years to become an expert at diagnosing complex problems. That expertise includes technical skills as well as best practices. The technical skills are what take a long time to learn and require experience and a lot of knowledge. The best practices, however, can be learned in just a few minutes. Here are six best practices that will help when diagnosing complex problems:

1. Collect relevant information when the problem occurs.
2. Keep a log of what you've done and what you think the problem might be.
3. Be detailed and avoid qualitative information.
4. Challenge assumptions until they are proven.
5. Narrow the scope of the problem.
6. Work to prove or disprove theories about the problem.

The best practices listed here are particularly important for complex problems that take a long time to solve. The more complex a problem is, the more important these best practices become. Each of the best practices is covered in more detail as follows.

1.3.3.1 Best Practices for Complex Investigations

1.3.3.1.1 Collect the Relevant Information When the Problem Occurs Earlier in this chapter we discussed how changes can cause certain types of problems. We also discussed how changes can remove evidence for why a problem occurred in the first place (for example, changes to the amount of free memory can hide the fact that it was once low). In the former situation, it is important to collect information because it can be compared to information that was collected at a previous time to see if any changes caused the problem. In the latter situation, it is important to collect information before the changes on the system wipe out any important evidence. The longer it takes to resolve a problem, the better the chance that something important will change during the investigation. In either situation, data collection is very important for complex problems.

Even reproducible problems can be affected by a changing system. A problem that occurs one day can stop occurring the next day because of an unknown change to the system. If you're lucky, the problem will never occur again, but that's not always the case.

Consider a problem that occurred many years ago where application trap occurred in one xterm (a type of terminal window) window but not in another. Both xterm windows were on the same system and were identical in every way (well, so it seemed at first) but still the problem occurred only in one. Even the list of environment variables was the same except for the expected differences such as PWD (present working directory). After logging out and back in, the problem could not be reproduced. A few days later the problem came back again, only in one xterm. After a very complex investigation, it turned out that an environment variable PWD was the difference that caused the problem to occur. This isn't as simple as it sounds. The contents of the PWD environment variable was not the cause of the problem, although the difference in size of PWD variables between the two xterms forced the stack (a special memory segment) to slightly move up or down in the address space. Sure enough, changing PWD to another value made the problem disappear or recur depending on the length. This small difference caused the different behavior for the application in the two xterms. In one xterm, a memory corruption in the application landed without issue on an inert part of the stack, causing no side-effect. In the other xterm, the memory corruption landed on a pointer on the stack (the long description of the problem is beyond the scope of this chapter). The pointer was dereferenced by the application, and the trap occurred. This is a very rare problem but is a good example of how small and seemingly unrelated changes or differences can affect a problem.

If the problem is serious and difficult to reproduce, collect and/or write down the information from 1.3.1: Initial Investigation Using Your Own Skills.

For quick reference, here is the list:

☞ The exact time the problem occurred
☞ Dynamic operating system information
☞ What you were doing when the problem occurred
☞ A problem description
☞ Anything that may have triggered the problem
☞ Any evidence that may be relevant

The more serious and complex the problem is, the more you'll want to start writing things down. With a complex problem, other people may need to get involved, and the investigation may get complex enough that you'll start to forget some of the information and theories you're using. The data collector included with this book can make your life easier whenever you need to collect information about the OS.

1.3.3.1.2 Use an Investigation Log Even if you only ever have one complex, critical problem to work on at a time, it is still important to keep track of what you've done. This doesn't mean well written, grammatically correct explanations of everything you've done, but it does mean enough detail to be useful to you at a later date. Assuming that you're like most people, you won't have the luxury of working on a single problem at a time, which makes this even more important. When you're investigating 10 problems at once, it sometimes gets difficult to keep track of what has been done for each of them. You also stand a good chance of hitting a similar problem again in the future and may want to use some of the information from the first investigation.

Further, if you ever need to get someone else involved in the investigation, an investigation log can prevent a great deal of unnecessary work. You don't want others unknowingly spending precious time re-doing your hard earned steps and finding the same results. An investigation log can also point others to what you have done so that they can make sure your conclusions are correct up to a certain point in the investigation.

An investigation log is a history of what has been done so far for the investigation of a problem. It should include theories about what the problem could be or what avenues of investigation might help to narrow down the problem. As much as possible, it should contain real evidence that helps lead you to the current point of investigation. Be very careful about making assumptions, and be very careful about qualitative proofs (proofs that contain no concrete evidence).

The following example shows a very structured and well laid out investigation log. With some experience, you'll find the format that works best for you. As you read through it, it should be obvious how useful an investigation log is. If you had to take over this problem investigation right now, it should be clear what has been done and where the investigator left off.

```
Time of occurrence: Sun Sep 5 21:23:58 EDT 2004
Problem description: Product Y failed to start when run from a cron
job.
Symptom:

ProdY: Could not create communication semaphore: 1176688244 (EEXIST)

What might have caused the problem: The error message seems to indicate
that the semaphore already existed and could not be recreated.
```

Theory #1: Product Y may have crashed abruptly, leaving one or more IPC resources. On restart, the product may have tried to recreate a semaphore that it already created from a previous run.

Needed to prove/disprove:
☞ The ownership of the semaphore resource at the time of the error is the same as the user that ran product Y.
☞ That there was a previous crash for product Y that would have left the IPC resources allocated.

Proof: Unfortunately, there was no information collected at the time of the error, so we will never truly know the owner of the semaphore at the time of the error. There is no sign of a trap, and product Y always leaves a debug file when it traps. This is an unlikely theory that is good given we don't have the information required to make progress on it.

Theory #2: Product X may have been running at the time, and there may have been an IPC (Inter Process Communication) key collision with product Y.

Needed to prove/disprove:
☞ Check whether product X and product Y can use the same IPC key.
☞ Confirm that both product X and product Y were actually running at the time.

Proof: Started product X and then tried to start product Y. Ran "strace" on product X and got the following semget:

```
ion 618% strace -o productX.strace prodX
ion 619% egrep "sem|shm" productX.strace
semget(1176688244, 1, 0)        = 399278084
```

```
Ran "strace" on product Y and got the following semget:

ion 730% strace -o productY.strace prodY
ion 731% egrep "sem|shm" productY.strace
semget(1176688244, 1, IPC_CREAT|IPC_EXCL|0x1f7|0666) = EEXIST

The IPC keys are identical, and product Y tries to create the semaphore
but fails. The error message from product Y is identical to the original
error message in the problem description here.

Notes: productX.strace and productY.strace are under the data directory.

Assumption: I still don't know whether product X was running at the
time when product Y failed to start, but given these results, it is very
likely. IPC collisions are rare, and we know that product X and product
Y cannot run at the same time the way they are currently configured.
```

Note: A semaphore is a special type of inter-process communication mechanism that provides a synchronization mechanism between processes (and/or threads). The type of semaphore used here requires a unique "key" so that multiple processes can use the same semaphore. A semaphore can exist without any processes using it, and some applications expect and rely on creating a semaphore before they can run properly. The `semget()` in the strace that follows is a system call (a special type of OS function) that, as the name suggests, gets a semaphore.

Notice how detailed the proofs are. Even the commands used to capture the original strace output are included to eliminate any human error. When entering a proof, be sure to ask yourself, "Would someone else need any more proof than this?" This level of detail is often required for complex problems so that others will see the proof and agree with it.

The amount of detail in your investigation log should depend on how critical the problem is and how close you are to solving it. If you're completely lost on a very critical problem, you should include more detail than if you are almost done with the investigation. The high level of detail is very useful for complex problems given that every piece of data could be invaluable later on in the investigation.

If you don't have a good problem tracking system, here is a possible directory structure that can help keep things organized:

```
<problem identifier>/ inv.txt
                    / data /
                    / src /
```

The problem identifier is for tracking purposes. Use whatever is appropriate for you (even if it is 1, 2, 3, 4, and so on). The inv.txt is the investigation log, containing the various theories and proofs. The data directory is for any data files that have been collected. Having one data directory helps keep things organized and it also makes it easy to refer to data files from your investigation log. The src directory is for any source code or scripts that you write to help investigate the problem.

The problem directory is what you would show someone when referring to the problem you are investigating. The investigation log would contain the flow of the investigation with the detailed proofs and should be enough to get someone up to speed quickly.

You may also want to save the problem directory for the future or better yet, put the investigation directories somewhere where others can search through them as well. After all, you worked hard for the information in your investigation log; don't be too quick to delete it. You never know when you'll hit a similar (or the same) problem again. The investigation log can also be used to help educate more junior people about investigation techniques.

1.3.3.1.3 Be Detailed (Avoid Qualitative Information) Be very detailed in your investigation log or any time when discussing the problem. If you prove a theory using an error record from an error log file, include the error record and the name of the error log file as proof in the investigation log. Avoid qualitative proofs such as, "Found an error log that showed that the suspect product was running at the time." If you transfer a problem to another person, that person will want to see the actual error record to ensure that your assumption was correct. Also if the problem lasts long enough, you may actually start to second-guess yourself as well (which is actually a good thing) and may appreciate that quantitative proof (a proof with real data to back it up).

Another example of a qualitative proof is a relative term or description. Descriptions like "the file was very large" and "the CPU workload was high" will mean different things to different people. You need to include details for how large the file was (using the output of the `ls` command if possible) and how high the CPU workload was (using `uptime` or `top`). This will remove any uncertainty that others (or you) have about your theories and proofs for the investigation.

Similarly, when you are asked to review an investigation, be leery of any proof or absolute statement (for example, "I saw the amount of virtual memory drop to dangerous levels last night") without the required evidence (that is, a log record, output from a specific OS command, and so on). If you don't have the actual evidence, you'll never know whether a statement is true. This doesn't mean that you have to distrust everyone you work with to solve a problem but rather a realization that people make mistakes. A quick cut and paste of an error log file or the output from an actual command might be all the evidence you need to agree with a statement. Or you might find that the statement is based on an incorrect assumption.

1.3.3.1.4 Challenge Assumptions There is nothing like spending a week diagnosing a problem based on an assumption that was incorrect. Consider an example where a problem has been identified and a fix has been provided ... yet the problem happens again. There are two main possibilities here. The first is that the fix didn't address the problem. The second is that the fix is good, but you didn't actually get it onto the system (for the statistically inclined reader: yes there is a chance that the fix is bad *and* it didn't get on the system, but the chances are very slim). For critical problems, people have a tendency to jump to conclusions out of desperation to solve a problem quickly. If the group you're working with starts complaining about the bad fix, you should encourage them to challenge both possibilities. Challenge the assumption that the fix actually got onto the system. (Was it even built into the executable or library that was supposed to contain the fix?)

1.3.3.1.5 Narrow Down the Scope of the Problem Solution (that is, a complete IT solution) -level problem determination is difficult enough, but to make matters worse, each application or product in a solution usually requires a different set of skills and knowledge. Even following the trail of evidence can require deep skills for each application, which might mean getting a few experts involved. This is why it is so important to try and narrow down the scope of the problem for a solution level problem as quickly as possible.

Today's complex heterogeneous solutions can make simple problems very difficult to diagnose. Computer systems and the software that runs on them are integrated through networks and other mechanism(s) to work together to provide a solution. A simple problem, even one that has a clear error message, can become difficult given that the effect of the problem can ripple throughout a solution, causing seemingly unrelated symptoms. Consider the example in Figure 1.1.

Application A in a solution could return an error code because it failed to allocate memory (effect #1). On its own, this problem could be easy to diagnose. However, this in turn could cause application B to react and return an error of its own (effect #2). Application D may see this as an indication that application B is unavailable and may redirect its requests to a redundant application C (effect #3). Application E, which relies on application D and serves the end user, may experience a slowdown in performance (effect #4) since application D is no longer using the two redundant servers B and C. This in turn can cause an end user to experience the performance degradation (effect #5) and to phone up technical support (effect #6) because the performance is slower than usual.

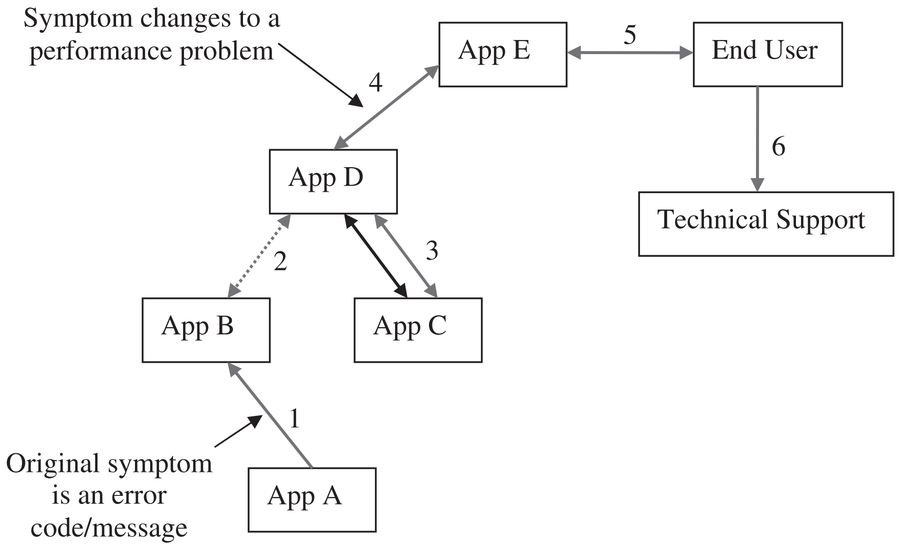

Symptom changes to a performance problem
App E
5
End User
4
6
App D
Technical Support
3
2
App B
App C
1
Original symptom is an error code/message
App A

Fig. 1.1 Ripple effect of an error in a solution.

If this seems overly complex, it is actually an oversimplification of real IT solutions where hundreds or even thousands of systems can be connected together. The challenge for the investigator is to follow the trail of evidence back to the original error.

It is particularly important to challenge assumptions when working on a solution-level problem. You need to find out whether each symptom is related to a local system or whether the symptom is related to a change or error condition in another part of a solution.

There are some complex problems that cannot be broken down in scope. These problems require true skill and perseverance to diagnose. Usually this type of problem is a race condition that is very difficult to reproduce. A *race condition* is a type of problem that depends on timing and the order in which things occur. A good example is a "late read." A late read is a software defect where memory is freed, but at some point in the very near future, it is used again by a different part of the application. As long as the memory hasn't been reused, the late read may be okay. However, if the memory block has been reused (and written to), the late read will access the new contents of the memory block, causing unpredictable behavior. Most race conditions can be narrowed in scope in one way or another, but some are so timing-dependent that any changes to the environment (for the purposes of investigation) will cause the problem to not occur.

Lastly, everyone working on an IT solution should be aware of the basic architecture of the solution. This will help the team narrow the scope of any problems that occur. Knowing the basic architecture will help people to theorize where a problem may be coming from and eventually identify the source.

1.3.3.2 Create a Reproducible Test Case Assuming you know how the problem occurs (note that the word here is *how*, not why), it will help others if you can create a test case and/or environment that can reproduce the problem at will. A *test case* is a term used to refer to a tool or a small set of commands that, when run, can cause a problem to occur.

A successful test case can greatly reduce the time to resolution for a problem. If you're investigating a problem on your own, you can run and rerun the test case to cause the problem to occur many times in a row, learning from the symptoms and using different investigation techniques to better understand the problem.

If you need to ask an expert for help, you will also get much more help if you include a reproducible test case. In many cases, an expert will know how to investigate a problem but not how to reproduce it. Having a reproducible test case is especially important if you are asking a stranger for help over the Internet. In this case, the person helping you will probably be doing so on his or her own time and will be more willing to help out if you make it as easy as you can.

1.3.3.3 Work to Prove and/or Disprove Theories This is part of any good problem investigation. The investigator will do his best to think of possible avenues of investigation and to prove or disprove them. The real art here is to identify theories that are easy to prove or disprove or that will dramatically narrow the scope of a problem.

Even nonsolution level problems (such as an application that fails when run from the command line) can be easier to diagnose if they are narrowed in scope with the right theory. Consider an application that is failing to start with an obscure error message. One theory could be that the application is unable to allocate memory. This theory is much smaller in scope and easier to investigate because it does not require intimate knowledge about the application. Because the theory is not application-specific, there are more people who understand how to investigate it. If you need to get an expert involved, you only need someone who understands how to investigate whether an application is unable to allocate memory. That expert may know nothing about the application itself (and might not need to).

1.3.3.4 The Source Code If you are familiar with reading C source code, looking at the source is always a great way of determining why something isn't working the way it should. Details of how and when to do this are discussed in several chapters of this book, along with how to make use of the cscope utility to quickly pinpoint specific source code areas.

Also included in the source code is the Documentation directory that contains a great deal of detailed documentation on various aspects of the Linux kernel in text files. For specific kernel related questions, performing a search command such as the following can quickly yield some help:

```
find /usr/src/linux/Documentation -type f |
xargs grep -H <search_pattern> | less
```

where **<search_pattern>** is the desired search criteria as documented in **grep(1)**.

1.3.4 Phase #4: Getting Help or New Ideas

Everyone gets stuck, and once you've looked at a problem for too long, it can be hard to view it from a different perspective. Regardless of whether you're asking a peer or an expert for ideas/help, they will certainly appreciate any homework you've done up to this point.

1.3.4.1 Profile of a Linux Guru A great deal of the key people working on Linux do so as a "side job" (which often receives more time and devotion than their regular full-time jobs). Many of these people were the original "Linux hackers" and are often considered the "Linux gurus" of today. It's important to understand that these Linux gurus spend a great deal of their own spare time working (sometimes affectionately called "hacking") on the Linux kernel. If they decide to help you, they will probably be doing so on their own time. That said, Linux gurus are a special breed of people who have great passion for the concept of open source, free software, and the operating system itself. They take the development and correct operation of the code very seriously and have great pride in it. Often they are willing to help if you ask the right questions and show some respect.

1.3.4.2 Effectively Asking for Help

1.3.4.2.1 Netiquitte *Netiquette* is a commonly used term that refers to Internet etiquette. Netiquette is all about being polite and showing respect to others on the Internet. One of the best and most succinct documents on netiquette is RFC1855 (RFC stands for "Request for Comments"). It can be found at `http://www.faqs.org/rfcs/rfc1855.html`. Here are a few key points from this document:

☞ Read both mailing lists and newsgroups for one to two months before you post anything. This helps you to get an understanding of the culture of the group.
☞ Consider that a large audience will see your posts. That may include your present or next boss. Take care in what you write. Remember too, that mailing lists and newsgroups are frequently archived and that your words may be stored for a very long time in a place to which many people have access.
☞ Messages and articles should be brief and to the point. Don't wander off-topic, don't ramble, and don't send mail or post messages solely to point out other people's errors in typing or spelling. These, more than any other behavior, mark you as an immature beginner.

Note that the first point tells you to read newsgroups and mailing lists for one to two months before you post anything. What if you have a problem now? Well, if you are responsible for supporting a critical system or a large group of users, don't wait until you need to post a message, starting getting familiar with the key mailing lists or newsgroups now.

Besides making people feel more comfortable about how you communicate over the Internet, why should you care so much about netiquette? Well, if you don't follow the rules of netiquette, people won't want to answer your requests for help. In other words, if you don't respect those you are asking for help, they aren't likely to help you. As mentioned before, many of the people who could help you would be doing so on their own time. Their motivation to help you is governed partially by whether you are someone they want to help. Your message or post is the only way they have to judge who you are.

There are many other Web sites that document common netiquette, and it is worthwhile to read some of these, especially when interacting with USENET and mailing lists. A quick search in Google will reveal many sites dedicated to netiquette. Read up!

1.3.4.2.2 Composing an Effective Message In this section we discuss how to create an effective message whether for email or for USENET. An effective message, as you can imagine, is about clarity and respect. This does not mean that you must be completely submissive — assertiveness is also important, but it is crucial to respect others and understand where they are coming from. For example, you will not get a very positive response if you post a message such as the following to a mailing list:

```
To: linux-kernel-mailing-list
From: Joe Blow
Subject: HELP NEEDED NOW: LINUX SYSTEM DOWN!!!!!!
Message:

MY LINUX SYSTEM IS DOWN!!!! I NEED SOMEONE TO FIX IT NOW!!!! WHY DOES
LINUX ALWAYS CRASH ON ME???!!!!

Joe Blow
Linux System Administrator
```

First of all, CAPS are considered an indication of yelling in current netiquette. Many people reading this will instantly take offense without even reading the complete message.

Second, it's important to understand that many people in the open source community have their own deadlines and stress (like everyone else). So when asking for help, indicating the severity of a problem is OK, but do not overdo it.

Third, bashing the product that you're asking help with is a very bad idea. The people who may be able to help you may take offense to such a comment. Sure, you might be stressed, but keep it to yourself.

Last, this request for help has no content to it at all. There is no indication of what the problem is, not even what kernel level is being used. The subject line is also horribly vague. Even respectful messages that do not contain any content are a complete waste of bandwidth. They will always require two more messages (emails or posts), one from someone asking for more detail (assuming that someone cares enough to ask) and one from you to include more detail.

Ok, we've seen an example of how *not* to compose a message. Let's reword that bad message into something that is far more appropriate:

```
To: linux-kernel-mailing-list
From: Joe Blow
Subject: Oops in zisofs_cleanup on 2.4.21
Message:

Hello All,
My Linux server has experienced the Oops shown below three times in
the last week while running my database management system. I have
tried to reproduce it, but it does not seem to be triggered by
anything easily executed. Has anyone seen anything like this before?

Unable to handle kernel paging request at virtual address
ffffffff7f1bb800
 printing rip:
ffffffff7f1bb800
PML4 103027 PGD 0
Oops: 0010
CPU 0
Pid: 7250, comm: foo Not tainted
```

```
RIP: 0010:[zisofs_cleanup+2132522656/-2146435424]
RIP: 0010:[<ffffffff7f1bb800>]
RSP: 0018:0000010059795f10 EFLAGS: 00010206
RAX: 0000000000000000 RBX: 0000010059794000 RCX: 0000000000000000
RDX: ffffffffffffffea RSI: 0000000000000018 RDI: 0000007fbfff8fa8
RBP: 00000000037e00de R08: 0000000000000000 R09: 0000000000000000
R10: 0000000000000000 R11: 0000000000000246 R12: 0000000000000009
R13: 0000000000000018 R14: 0000000000000018 R15: 0000000000000000
FS: 0000002a957819e0(0000) GS:ffffffff804beac0(0000)
knlGS:0000000000000000
CS: 0010 DS: 0000 ES: 0000 CR0: 000000008005003b
CR2: ffffffff7f1bb800 CR3: 0000000000101000 CR4: 00000000000006e0
Process foo (pid: 7250, stackpage=10059795000)
Stack: 0000010059795f10 0000000000000018 ffffffff801bc576
➥0000010059794000
   0000000293716a88 0000007fbfff8da0 0000002a9cf94ff8
➥0000000000000003
   0000000000000000 0000000000000000 0000007fbfff9d64
➥0000007fbfff8ed0
Call Trace: [sys_msgsnd+134/976]{sys_msgsnd+134} [system_call+119/
124]{system_call+119}
Call Trace: [<ffffffff801bc576>]{sys_msgsnd+134}
[<ffffffff801100b3>]{system_call+119}

Thanks in advance,
Joe Blow
```

The first thing to notice is that the subject is clear, concise, and to the point. The next thing to notice is that the message is polite, but not overly mushy. All necessary information is included such as what was running when the oops occurred, an attempt at reproducing was made, and the message includes the Oops Report itself. This is a good example because it's one where further analysis is difficult. This is why the main question in the message was if anyone has ever seen anything like it. This question will encourage the reader at the very least to scan the Oops Report. If the reader has seen something similar, there is a good chance that he or she will post a response or send you an email. The keys again are respect, clarity, conciseness, and focused information.

1.3.4.2.3 Giving Back to the Community The open source community relies on the sharing of knowledge. By searching the Internet for other experiences with the problem you are encountering, you are relying on that sharing. If the problem you experienced was a unique one and required some ingenuity either on your part or someone else who helped you, it is very important to give back to the community in the form of a follow-up message to a post you have made. I have come across many message threads in the past where someone posted a question that was exactly the same problem I was having. Thankfully, they responded to their own post and in some cases even prefixed the original subject with "SOLVED:" and detailed how they solved the problem. If that person had not taken the time to post the second message, I might still be looking for the answer to my question. Also think of it this way: By posting the answer to USENET, you're also very safely archiving information at no cost to you! You could attempt to save the information locally, but unless you take very good care, you may lose the info either by disaster or by simply misplacing it over time.

If someone responded to your plea for help and helped you out, it's always a very good idea to go out of your way to thank that person. Remember that many Linux gurus provide help on their own time and not as part of their regular jobs.

1.3.4.2.4 USENET When posting to USENET, common netiquette dictates to only post to a single newsgroup (or a very small set of newsgroups) and to make sure the newsgroup being posted to is the correct one. If the newsgroup is not the correct one, someone may forward your message if you're lucky; otherwise, it will just get ignored.

There are thousands of USENET newsgroups, so how do you know which one to post to? There are several Web sites that host lists of available newsgroups, but the problem is that many of them only list the newsgroups provided by a particular news server. At the time of writing, Google Groups 2 (`http://groups-beta.google.com/`) is currently in beta and offers an enhanced interface to the USENET archives in addition to other group-based discussion archives. One key enhancement of Google Groups 2 is the ability to see all newsgroup names that match a query. For example, searching for "gcc" produces about half of a million hits, but the matched newsgroup names are listed before all the results. From this listing, you will be able to determine the most appropriate group to post a question to.

Of course, there are other resources beyond USENET you can send a message to. You or your company may have a support contract with a distribution or consulting firm. In this case, sending an email using the same tips presented in this chapter still apply.

1.3.4.2.5 Mailing Lists As mentioned in the RFC, it is considered proper netiquette to not post a question to a mailing list without monitoring the emails for a month or two first. Active subscribers prefer users to *lurk* for a while before posting a question. The act of lurking is to subscribe and read incoming posts from other subscribers without posting anything of your own.

An alternative to posting a message to a newsgroup or mailing list is to open a new bug report in a Bugzilla database, if one exists for the package in question.

1.3.4.2.6 Tips on Opening Bug Reports in Bugzilla When you open a bug report in Bugzilla, you are asking someone else to look into the problem for you. Any time you transfer a problem to someone else or ask someone to help with a problem, you need to have clear and concise information about the problem. This is common sense, and the information collected in Phase #3 will pretty much cover what is needed. In addition to this, there are some Bugzilla specific pointers, as follows:

- ☞ Be sure to properly characterize the bug in the various drop-down menus of the bug report screen. See as an example the new bug form for GCC's Bugzilla, shown in Figure 1.2. It is important to choose the proper version and component because components in Bugzilla have individual owners who get notified immediately when a new bug is opened against their components.

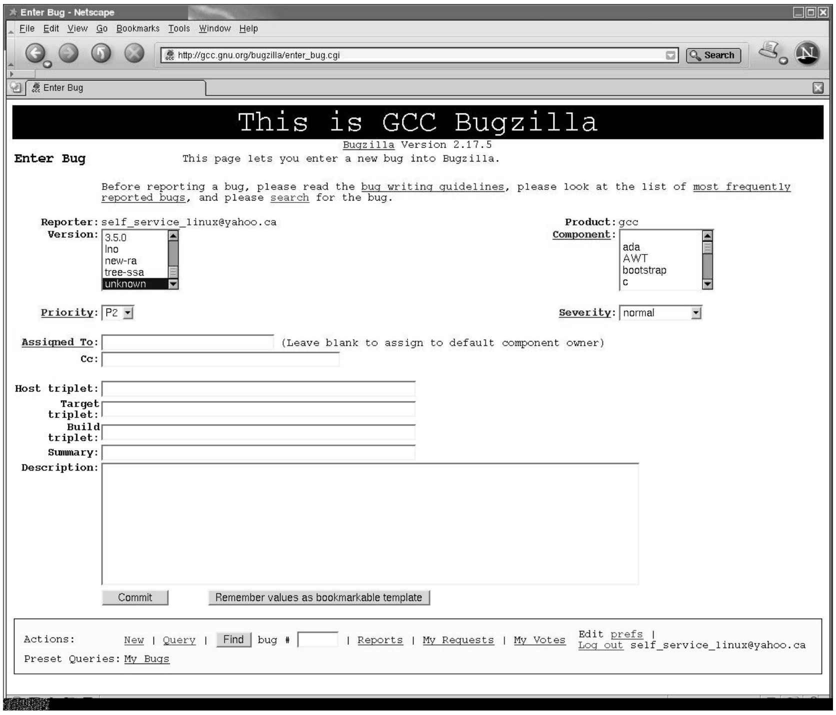
- ☞ Enter a clear and concise summary into the *Summary* field. This is the first and sometimes only part of a bug report that people will look at, so it is crucial to be clear. For example, entering `Compile aborts` is very bad. Ask yourself the same questions others would ask when reading this summary: "How does it break?" "What error message is displayed?" and "What kind of compile breaks?" A summary of `gcc -c foo.c -O3 for gcc3.4 throws sigsegv` is much more meaningful. (Make it a part of your *lurking* to get a feel for how bug reports are usually built and model yours accordingly.)

- ☞ In the *Description* field, be sure to enter a clear report of the bug with as much information as possible. Namely, the following information should be included for all bug reports:

  - ☞ Exact version of the software being used
  - ☞ Linux distribution being used
  - ☞ Kernel version as reported by `uname -a`
  - ☞ How to easily reproduce the problem (if possible)
  - ☞ Actual results you see - cut and paste output if possible
  - ☞ Expected results - detail what you expect to see

- ☞ Often Bugzilla databases include a feature to attach files to a bug report. If this is supported, attach any files that you feel are necessary to help the developers reproduce the problem. See Figure 1.2.

> **Note**: The ability for others to reproduce the problem is crucial. If you cannot easily reproduce the bug, it is unlikely that a developer will investigate it beyond speculating what the problem may be based on other known problems.


Enter Bug - Netscape
File Edit View Go Bookmarks Tools Window Help
http://gcc.gnu.org/bugzilla/enter_bug.cgi
Search
Enter Bug

This is GCC Bugzilla

Bugzilla Version 2.17.5

Enter Bug This page lets you enter a new bug into Bugzilla.

Before reporting a bug, please read the bug writing guidelines, please look at the list of most frequently reported bugs, and please search for the bug.

Reporter:	self_service_linux@yahoo.ca	Product:	gcc
Version:	3.5.0, lno, new-ra, tree-ssa, unknown	Component:	ada, AWT, bootstrap, c
Priority:	P2	Severity:	normal
Assigned To:	(Leave blank to assign to default component owner)		
Cc:			
Host triplet:			
Target triplet:			
Build triplet:			
Summary:			
Description:			

Commit | Remember values as bookmarkable template

Actions: New | Query | Find bug # | Reports | My Requests | My Votes Edit prefs | Log out self_service_linux@yahoo.ca

Preset Queries: My Bugs

Fig. 1.2 Bugzilla

1.3.4.3 Use Your Distribution's Support If you or your business has purchased a Linux distribution from one of the distribution companies such as Novell/SuSE, Redhat, or Mandrake, it is likely that some sort of support offering is in place. Use it! That's what it is there for. It is still important, though, to do some homework on your own. As mentioned before, it can be faster than simply asking for help at the first sign of trouble, and you are likely to pick up some knowledge along the way. Also any work you do will help your distribution's support staff solve your problem faster.

1.4 Technical Investigation

The first section of this chapter introduced some good investigation practices. Good investigation practices lay the groundwork for efficient problem investigation and resolution, but there is still obviously a technical aspect to diagnosing problems. The second part of this chapter covers a technical overview for how to investigate common types of problems. It highlights the various types of problems and points to the more in-depth documentation that makes up the remainder of this book.

1.4.1 Symptom Versus Cause

Symptoms are the external indications that a problem occurred. The symptoms can be a hint to the underlying cause, but they can also be misleading. For example, a memory leak can manifest itself in many ways. If a process fails to allocate memory, the symptom could be an error message. If the program does not check for out of memory errors, the lack of memory could cause a trap (SIGSEGV). If there is *not* enough memory to log an error message, it could result in a trap because the kernel may be unable to grow the stack (that is, to call the error logging function). A memory leak could also be noticed as a growing memory footprint. A memory leak can have many symptoms, although regardless of the symptom, the cause is still the same.

Problem investigations always start with a symptom. There are five categories of symptoms listed below, each of which has its own methods of investigation.

1. Error
2. Crash
3. Hang (or very slow performance)
4. Performance
5. Unexpected behavior/output

1.4.1.1 Error Errors (and/or warnings) are the most frequent symptoms. They come in many forms and occur for many reasons including configuration issues, operating system resource limitations, hardware, and unexpected situations. Software produces an error message when it can’t run as expected. Your job as a problem investigator is to find out why it can’t run as expected and solve the underlying problem.

Error messages can be printed to the terminal, returned to a Web browser or logged to an error log file. A program usually uses what is most convenient and useful to the end user. A command line program will print error messages to the terminal, and a background process (one that runs without a command line) usually uses a log file. Regardless of how and where an error is produced, Figure 1.3 shows some of the initial and most useful paths of investigation for errors.

Unfortunately, errors are often accompanied by error messages that are not clear and do not include associated actions. Application errors can occur in obscure code paths that are not exercised frequently and in code paths where the full impact (and reason) for the error condition is not known. For example, an error message may come from the failure to open a file, but the purpose of opening a file might have been to read the configuration for an application. An error message of "could not open file" may be reported at the point where the error occurred and may not include any context for the severity, purpose, or potential action to solve the problem. This is where the strace and ltrace tools can help out.

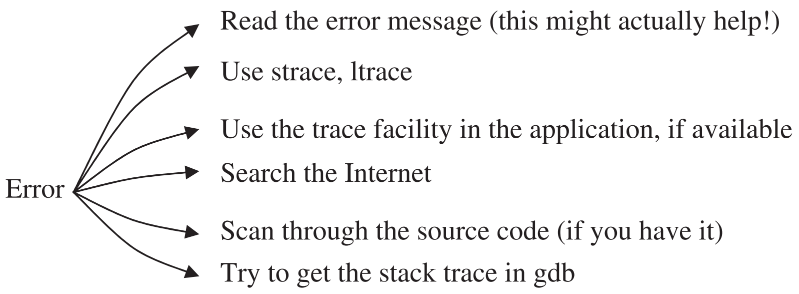

Error
Read the error message (this might actually help!)
Use strace, ltrace
Use the trace facility in the application, if available
Search the Internet
Scan through the source code (if you have it)
Try to get the stack trace in gdb

Fig. 1.3 Basic investigation for error symptoms.

Many types of errors are related to the operating system, and there is no better tool than strace to diagnose these types of errors. Look for system calls (in the strace output) that have failed right before the error message is printed to the terminal or logged to a file. You might see the error message printed via the `write()` system call. This is the system call that `printf`, `perror`, and other print-like functions use to print to the terminal. Usually the failing system call is very close to where the error message is printed out. If you need more information than what strace provides, it might be worthwhile to use the ltrace tool (it is similar to the strace tool but includes function calls). For more information on strace, refer to Chapter 2.

If strace and ltrace utilities do not help identify the problem, try searching the Internet using the error message and possibly some key words. With so many Linux users on the Internet, there is a chance that someone has faced the problem before. If they have, they may have posted the error message and a solution. If you run into an error message that takes a considerable amount of time to resolve, it might be worthwhile (and polite) to post a note on USENET with the original error message, any other relevant information, and the resulting solution. That way, if someone hits the same problem in the future, they won't have to spend as much time diagnosing the same problem as you did.

If you need to dig deeper (strace, ltrace, and the Internet can't help), the investigation will become very specific to the application. If you have source code, you can pinpoint where the problem occurred by searching for the error message directly in the source code. Some applications use error codes and not raw error messages. In this case, simply look for the error message, identify the associated error code, and search for it in source code. If the same error code/message is used in multiple places, it may be worthwhile to add a `printf()` call to differentiate between them.

If the error message is unclear, strace and ltrace couldn't help, the Internet didn't have any useful information, and you don't have the source code, you still might be able to make further progress with GDB. If you can capture the point in time in GDB when the application produces the error message, the functions on the stack may give you a hint about the cause of the problem. This won't be easy to do. You might have to use break points on the `write()` system call and check whether the error message is being written out. For more information on how to use GDB, refer to Chapter 6, "The GNU Debugger (GDB)."

If all else fails, you'll need to contact the support organization for the application and ask them to help with the investigation.

1.4.1.2 Crashes Crashes occur because of severe conditions and fit into two main categories: traps and panics. A trap usually occurs when an application references memory incorrectly, when a bad instruction is executed, or when there is a bad "page-in" (the process of bringing a page from the swap area into memory). A panic in an application is due to the application itself abruptly shutting down due to a severe error condition. The main difference is that a trap is a crash that the hardware and OS initiate, and a panic is a crash that the application initiates. Panics are usually associated with an error message that is produced prior to the panic. Applications on Unix and Linux often panic by calling the `abort()` function (after the error message is logged or printed to the terminal).

Like errors, crashes (traps and panics) can occur for many reasons. Some of the more popular are included in Figure 1.4.

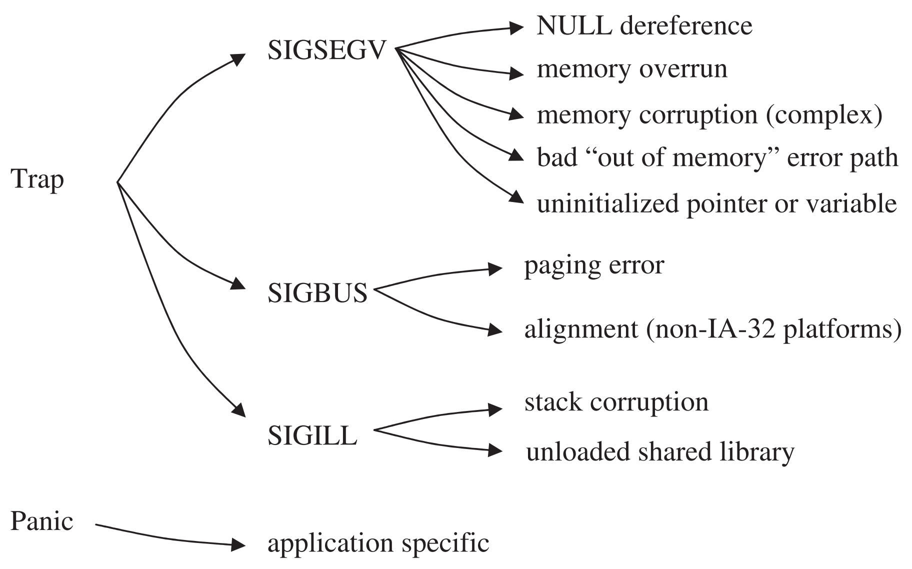


Fig. 1.4 Common causes of crashes.

1.4.1.2.1 Traps When the kernel experiences a major problem while running a process, it may send a signal (a Unix and Linux convention) to the process such as SIGSEGV, SIGBUS or SIGILL. Some of these signals are due to a hardware condition such as an attempt to write to a write-protected region of memory (the kernel gets the actual trap in this case). Other signals may be sent by the kernel because of non-hardware related issues. For example, a bad page-in can be caused by a failure to read from the file system.

The most important information to gather for a trap is:

☞ **The instruction that trapped.** The instruction can tell you a lot about the type of trap. If the instruction is invalid, it will generate a SIGILL. If the instruction references memory and the trap is a SIGSEGV, the trap is likely due to referencing memory that is outside of a memory region (see Chapter 3 on the /proc file system for information on process memory maps).

☞ **The function name and offset of the instruction that trapped.** This can be obtained through GDB or using the load address of the shared library and the instruction address itself. More information on this can be found in Chapter 9, "ELF: Executable Linking Format."

☞ **The stack trace.** The stack trace can help you understand why the trap occurred. The functions that are higher on the stack may have passed a bad pointer to the lower functions causing a trap. A stack trace can also be used to recognize known types of traps. For more information on stack trace backs refer to Chapter 5, "The Stack."

☞ **The register dump.** The register dump can help you understand the "context" under which the trap occurred. The values of the registers may be required to understand what led up to the trap.

☞ **A core file or memory dump.** This can fill in the gaps for complex trap investigations. If some memory was corrupted, you might want to see how it was corrupted or look for pointers into that area of corruption. A core file or memory dump can be very useful, but it can also be very large. For example, a 64-bit application can easily use 20GB of memory or more. A full core file from such an application would be 20GB in size. That requires a lot of disk storage and may need to be transferred to you if the problem occurred on a remote and inaccessible system (for example, a customer system).

Some applications use a special function called a "signal handler" to generate information about a trap that occurred. Other applications simply trap and die immediately, in which case the best way to diagnose the problem is through a debugger such as GDB. Either way, the same information should be collected (in the latter case, you need to use GDB).

A SIGSEGV is the most common of the three *bad programming signals*: SIGSEGV, SIGBUS and SIGILL. A bad programming signal is sent by the kernel and is usually caused by memory corruption (for example, an overrun), bad memory management (that is, a duplicate free), a bad pointer, or an uninitialized value. If you have the source code for the tool or application and some knowledge of C/C++, you can diagnose the problem on your own (with some work). If you don't have the source code, you need to know assembly language to properly diagnose the problem. Without source code, it will be a real challenge to fix the problem once you've diagnosed it.

For memory corruption, you might be able to pinpoint the stack trace that is causing the corruption by using *watch points* through GDB. A watch point is a special feature in GDB that is supported by the underlying hardware. It allows you to stop the process any time a range of memory is changed. Once you know the address of the corruption, all you have to do is recreate the problem under the same conditions with a watch point on the address that gets corrupted. More on watch points in the GDB chapter.

There are some things to check for that can help diagnose operating system or hardware related problems. If the memory corruption starts and/or ends on a page sized boundary (4KB on IA-32), it could be the underlying physical memory or the memory management layer in the kernel itself. Hardware-based corruption (quite rare) often occurs at cache line boundaries. Keep both of them in mind when you look at the type of corruption that is causing the trap.

The most frequent cause of a SIGBUS is misaligned data. This does not occur on IA-32 platforms because the underlying hardware silently handles the misaligned memory accesses. However on IA-32, a SIGBUS can still occur for a bad page fault (such as a bad page-in).

Another type of hardware problem is when the instructions just don't make sense. You've looked at the memory values, and you've looked at the registers, but there is no way that the instructions could have caused the values. For example, it may look like an increment instruction failed to execute or that a subtract instruction did not take place. These types of hardware problems are very rare but are also very difficult to diagnose from scratch. As a rule of thumb, if something looks impossible (according to the memory values, registers, or instructions), it might just be hardware related. For a more thorough diagnosis of a SIGSEGV or other traps, refer to Chapter 6.

1.4.1.2.2 Panics A panic in an application is due to the application itself abruptly shutting down. Linux even has a system call specially designed for this sort of thing: abort (although there are many other ways for an application to "panic"). A panic is a similar symptom to a trap but is much more purposeful. Some products might panic to prevent further risk to the users' data or simply because there is no way it can continue. Depending on the application, protecting the users' data may be more important than trying to continue running. If an application's main control block is corrupt, it might mean that the application has no choice but to panic and abruptly shutdown. Panics are very product-specific and often require knowledge of the product (and source code) to understand. The line number of the source code is sometimes included with a panic. If you have the source code, you might be able to use the line of code to figure out what happened.

Some panics include detailed messages for what happened and how to recover. This is similar to an error message except that the product (tool or application) aborted and shut down abruptly. The error message and other evidence of the panic usually have some good key words or sentences that can be searched for using the Internet. The panic message may even explain how to recover from the problem.

If the panic doesn't have a clear error message and you don't have the source code, you might have to ask the product vendor what happened and provide information as needed. Panics are somewhat rare, so hopefully you won't encounter them often.

1.4.1.2.3 Kernel Crashes A panic or trap in the kernel is similar to those in an application but obviously much more serious in that they often affect the entire system. Information for how to investigate system crashes and hangs is fairly complex and not covered here but is covered in detail in Chapter 7, "Linux System Crashes and Hangs."

1.4.1.3 Hangs (or Very Slow Performance) It is difficult to tell the difference between a hang and very slow performance. The symptoms are pretty much identical as are the initial methods to investigate them. When investigating a perceived hang, you need to find out whether the process is hung, looping, or performing very slowly. A true hang is when the process is not consuming any CPU and is stuck waiting on a system call. A process that is looping is consuming CPU and is usually, but not always, stuck in a tight code loop (that is, doing the same thing over and over). The quickest way to determine what type of hang you have is to collect a set of stack traces over a period of time and/or to use GDB and strace to see whether the process is making any progress at all. The basic investigation steps are included in Figure 1.5.

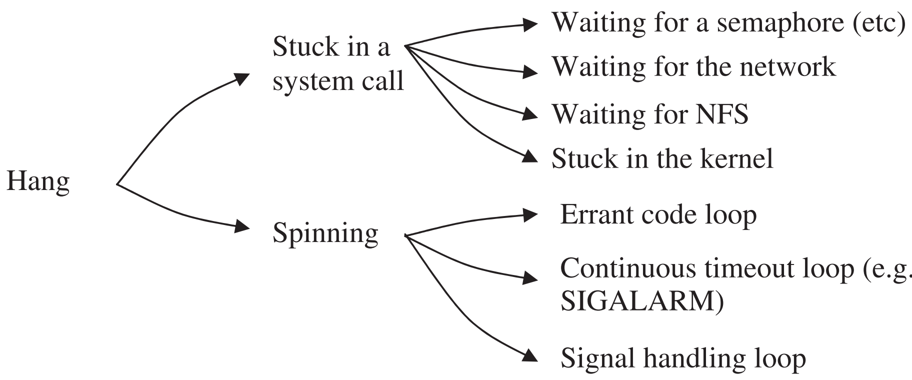


Fig. 1.5 Basic investigation steps for a hang.

If the application seems to be hanging, use GDB to get a stack trace (use the `bt` command). The stack trace will tell you where in the application the hang *may* be occurring. You still won't know whether the application is actually hung or whether it is looping. Use the `cont` command to let the process continue normally for a while and then stop it again with Control-C in GDB. Gather another stack trace. Do this a few times to ensure that you have a few stack traces over a period of time. If the stack traces are changing in any way, the process may be looping. However, there is still a chance that the process is making progress, albeit slowly. If the stack traces are identical, the process may still be looping, although it would have to be spending the majority of its time in a single state.

With the stack trace and the source code, you can get the line of code. From the line of code, you'll know what the process is waiting on but maybe not why. If the process is stuck in a semop (a system call that deals with semaphores), it is probably waiting for another process to notify it. The source code should explain what the process is waiting for and potentially what would wake it up. See Chapter 4, "Compiling," for information about turning a function name and function offset into a line of code.

If the process is stuck in a `read` call, it may be waiting for NFS. Check for NFS errors in the system log and use the mount command to help check whether any mount points are having problems. NFS problems are usually not due to a bug on the local system but rather a network problem or a problem with the NFS server.

If you can't attach a debugger to the hung process, the debugger hangs when you try, or you can't kill the process, the process is probably in some strange state in the kernel. In this rare case, you'll probably want to get a kernel stack for this process. A kernel stack is stack trace for a task (for example, a process) in the kernel. Every time a system call is invoked, the process or thread will run some code in the kernel, and this code creates a stack trace much like code run outside the kernel. A process that is stuck in a system call will have a stack trace in the kernel that may help to explain the problem in more detail. Refer to Chapter 8, "Kernel Debugging with KDB," for more information on how to get and interpret kernel stacks.

The strace tool can also help you understand the cause of a hang. In particular, it will show you any interaction with the operating system. However, strace will not help if the process is spinning in user code and never calls a system call. For signal handling loops, the strace tool will show very obvious symptoms of a repeated signal being generated and caught. Refer to the hang investigation in the strace chapter for more information on how to use strace to diagnose a hang with strace.

1.4.1.3.1 Multi-Process Applications For multi-process applications, a hang can be very complex. One of the processes of the application could be causing the hang, and the rest might be hanging waiting for the hung process to finish. You'll need to get a stack trace for all of the processes of the application to understand which are hung and which are causing the hang.

If one of the processes is hanging, there may be quite a few other processes that have the same (or similar) stack trace, all waiting for a resource or lock held by the original hung process. Look for a process that is stuck on something unique, one that has a unique stack trace. A unique stack trace will be different than all the rest. It will likely show that the process is stuck waiting for a reason of its own (such as waiting for information from over the network).

Another cause of an application hang is a dead lock/latch. In this case, the stack traces can help to figure out which locks/latches are being held by finding the source code and understanding what the source code is waiting for. Once you know which locks or latches the processes are waiting for, you can use the source code and the rest of the stack traces to understand where and how these locks or latches are acquired.

> **Note**: A *latch* usually refers to a very light weight locking mechanism. A lock is a more general term used to describe a method to ensure mutual exclusion over the access of a resource.

1.4.1.3.2 Very Busy Systems Have you ever encountered a system that seems completely hung at first, but after a few seconds or minutes you get a bit of response from the command line? This usually occurs in a terminal window or on the console where your key strokes only take effect every few seconds or longer. This is the sign of a very busy system. It could be due to an overloaded CPU or in some cases a very busy disk drive. For busy disks, the prompt may be responsive until you type a command (which in turn uses the file system and the underlying busy disk).

The biggest challenge with a problem like this is that once it occurs, it can take minutes or longer to run any command and see the results. This makes it very difficult to diagnose the problem quickly. If you are managing a small number of systems, you might be able to leave a special telnet connection to the system for when the problem occurs again.

The first step is to log on to the system before the problem occurs. You'll need a root account to renice (reprioritize) the shell to the highest priority, and you should change your current directory to a file system such as /proc that does not use any physical disks. Next, be sure to unset the LD_LIBRARY_PATH and PATH environment variables so that the shell does not search for libraries or executables. Also when the problem occurs, it may help to type your commands into a separate text editor (on another system) and paste the entire line into the remote (for example, telnet) session of the problematic system.

When you have a more responsive shell prompt, the normal set of commands (starting with `top`) will help you to diagnose the problem much faster than before.

1.4.1.4 Performance Ah, performance ... one could write an entire book on performance investigations. The quest to improve performance comes with good reason. Businesses and individuals pay good money for their hardware and are always trying to make the most of it. A 15% improvement in performance can be worth 15% of your hardware investment.

Whatever the reason, the quest for better performance will continue to be important. Keep in mind, however, that it may be more cost effective to buy a new system than to get that last 10-20%. When you're trying to get that last 10-20%, the human cost of improving performance can outweigh the cost of purchasing new hardware in a business environment.

1.4.1.5 Unexpected Behavior/Output This is a special type of problem where the application is not aware of a problem (that is, the error code paths have not been triggered), and yet it is returning incorrect information or behaving incorrectly. A good example of unexpected output is if an application returned "!$#%#@" for the current balance of a bank account without producing any error messages. The application may not execute any error paths at all, and yet the resulting output is complete nonsense. This type of problem can be difficult to diagnose given that the application will probably not log any diagnostic information (because it is not aware there is a problem!).

Note: An error path is a special piece of code that is specifically designed to react and handle an error.

The root cause for this type of problem can include hardware issues, memory corruptions, uninitialized memory, or a software bug causing a variable overflow. If you have the output from the unexpected behavior, try searching the Internet for some clues. Failing that, you're probably in for a complex problem investigation.

Diagnosing this type of problem manually is a lot easier with source code and an understanding of how the code is supposed to work. If the problem is easily reproducible, you can use GDB to find out where the unexpected behavior occurs (by using break points, for example) and then backtracking through many iterations until you've found where the erroneous behavior starts. Another option if you have the source code is to use printf statements (or something similar) to backtrack through the run of the application in the hopes of finding out where the incorrect behavior started.

You can try your luck with strace or ltrace in the hopes that the application is misbehaving due to an error path (for example, a file not found). In that particular case, you might be able to address the reason for the error (that is, fix the permissions on a file) and avoid the error path altogether.

If all else fails, try to get a subject matter expert involved, someone who knows the application well and has access to source code. They will have a better understanding of how the application works internally and will have better luck understanding what is going wrong. For commercial software products, this usually means contacting the software vendor for support.

1.5 Troubleshooting Commercial Products

In today's ever growing enterprise market, Linux is making a very real impact. A key to this impact is the availability of large scale software products such as database management systems, Web servers, and business solutions systems. As more companies begin to examine their information technology resources and spending, it is inevitable that they will at the very least consider using Linux in their environments. Even though there is a plethora of open source software available, many companies will still look to commercial software to provide a specific service or need.

With a rich problem determination and debugging skill set in-house, many problems that typically go to a commercial software vendor for support could be solved much faster internally. The intention of this book is to increase that skill set and give developers, support staff, or anyone interested the right toolkit to confidently tackle these problems. Even if the problem does in fact lie in the commercial software product, having excellent problem determination skills will greatly expedite the whole process of communicating and isolating the problem. This can mean differences of days or weeks of working with commercial software support staff.

It is also extremely important to read the commercial software's documentation, in particular, sections that discuss debugging and troubleshooting. Any large commercial application will include utilities and built-in problem determination facilities. These can include but are certainly not limited to:

- Dump files produced at the time of a trap (or set of predefined signals) that include information such as:
  - a stack traceback
  - contents of system registers at the time the signal was received
  - operating system/kernel information
  - process ID information
  - memory dumps of the software's key internal data structures
  - and so on
- Execution tracing facilities

- ☞ Diagnostic log file(s)
- ☞ Executables to examine and dump internal structures

Becoming familiar with a commercial product's included problem determination facilities along with what Linux offers can be a very solid defense against any software problem that may arise.

1.6 Conclusion

The rest of the book goes into much more detail, each chapter exploring intimate details of problem diagnosis with the available tools on Linux. Each chapter is designed to be practical and still cover some of the background information required to build deep skills.