Application Troubleshooting: Alternate Methods of Debugging

By Chris Duncan - Sun High End Services (HES) CTE-HPC

Sun BluePrints™ OnLine - November 2001

http://www.sun.com/blueprints

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Part No.: 816-1941-10
Revision 1.0, 10/08/01
Edition: November 2001

Please
Recycle

Application Troubleshooting: Alternate Methods of Debugging

Application failures and anomalies are a critical issue for everyone involved: users, system administrators, support personnel, and developers. This article shows the reader a variety of tools available to a Solaris™ Operating Environment (Solaris OE) user for working with and debugging problem applications. These tools provide alternate methods of debugging to be used instead of, or in addition to, a source code level debugger.

Some of these tools are quite easy to use. With an advanced knowledge of UNIX® platforms and C programming, these utilities can become very effective for pinpointing the failure area. They can be especially useful on applications which have not crashed, but are still running. At a minimum, these utilities will provide information about the failure and indicate areas for further investigation, either by source code level debugging, or as information to be passed on to support personnel. Although application users and system administrators will find these techniques useful, even the seasoned source code developer will find these techniques and utilities augment their arsenal.

Learning New Debugging Methods

One suggestion when working with new methods or utilities is to first try them with a simple example code. A short program that contains a few functions and contains some behaviors you wish to explore can be used to great advantage. Even a small C program that de-references a bad pointer or tries to open the wrong file can help you learn a lot about the utilities discussed here. By using a small example you can concentrate on a few simple behaviors and avoid the complexities in a typical application.

For most of the utilities discussed here, their man pages (the standard UNIX documentation available through the `man(1)` command) are a good source for information. Though they do not always have many direct output examples, they do have the typical overview of options and a sparse number of command line examples.

Solaris OE and UNIX Utilities

`/bin/truss`

The first utility discussed is likely the most powerful and rich in features: `truss`. This utility is a veritable swiss army knife for watching a program's execution. It can follow a running application and report on a variety of its system-level and internal activities without the need for the application's source code. `truss`'s main features include:

- Trace an application's execution at the user-level function and system call level.
- Follow threads and processes that `fork()` from the starting process.
- Run the application directly or attach to a running process.
- Stop the application's execution if certain events occur, such as a signal, system call, or user-level function call.
- Show the contents of the I/O buffers of `write()` and `read()` calls.
- Release traced processes when `truss` receives a SIGHUP, SIGINT, or SIGQUIT.

By default, `truss` follows all system calls (such as `open()`, `fork()`, `malloc()`, `read()`, and `write()`). Thus, the default output gives the user a large amount of information from which to peruse from a failing application. The examples that follow will cover some of the most useful command combinations.

One of the more bothersome debugging problems occur when applications open multiple files and do not handle an open failure gracefully. This can lead to the application dying a quick and possibly quiet death. The first example shows how to find all the files that a program, in this case called `a.out`, attempts to open:

```
% truss -t \!all -t open a.out
```

This example is particularly useful when an application exits abruptly with an obtuse or unclear error message. By following the application with `truss` the user can find out what was going on just before the application exited. For example, it's possible a configuration directory was moved or deleted by accident, and the application is not robust enough to detect the problem and take corrective action.

A particularly difficult class of problems to debug is when an application "hangs." Unfortunately "hang" is far from a technical term. It often merely means that the program has not completed execution long after its expected time, or is not generating output. For this case, there are several possible issues which are quite diverse. The program could be waiting for I/O from a device like a disk, or from a network socket connected to some other program. Multithreaded applications can sometimes "hang" when mutex locks are not properly handled, or the code has race condition bugs. There is also the unfortunate case when there is no technical problem at all - the code is running fine, but might be behind schedule due to heavy machine usage or the user not having a good handle on its completion time. In these cases where the application appears to be "hung," one can attach `truss` to see the program's system level interactions:

```
% truss -lfp process-id
```

This will attach `truss` to the processes listed on the command line, and output the system calls made by the application. This output may show that the application is continuously trying to open or write to a file and failing. There may be a case where it is sitting in a `sigsuspend()` or `poll()` loop waiting on an open socket or other input. This could lead to looking at other applications that are connected to the program.

Unfortunately, sometimes the output of a `truss` attach, as listed in the previous example, is inconclusive—the output may even be very close to what is output by `truss` when the application appears to run fine. In this case, one could further enhance the output and delve into internal user-level functions that are being called by the application (Note that user-level function tracing is available in Solaris 8 OE):

```
% truss -t \!all -fl -u a.out -p process-id
```

Sometimes, depending on the coding style and function naming scheme, the order of the functions called may help in determining what is happening. However, a user may unfortunately be "hung" inside a single user-level function, which will cause the previous command to have no output. In the case of a user mistakenly thinking the program was hung, it could be in a large loop doing numerical calculations or such. In this last case, check to see if the process is continuing to use CPU time by using the `ps` command.

There may also be cases where programs fail and then attempt to do cleanup, which is nice for saving file space, but may destroy all evidence useful for debugging purposes. The following is an example of using `truss` to execute a program `a.out`, and stop it if it incurs a SIGSEGV or SIGBUS:

```
% truss -f -t \!all -S SEGV,BUS a.out
```

One caveat is that `truss` will leave the process in the background, and exit in this case, thereby returning the user to the shell prompt. Since the user supplied the `-f` option, they know the process id of the program, and so can attach to it with a debugger or use other utilities.

Lastly, `truss` will leave the process truly stopped in the sense that it will no longer be scheduled to run. The program will not process signals such as SIGCONT or SIGTERM, and the user will have to "`kill -9`" it, or use a utility called `prun`.

There are many other ways to use `truss`, including detecting machine faults (not to be confused with general hardware errors or failures), counting the total numbers of system calls made, getting timestamps of each event, finding the environment passed to a child process, and other important information. The `truss` man page has quite a long list of options, examples, and information.

`truss` is a excellent way to follow an application's interactions with the system, and can even allow some of its internal workings to be followed. This functionality is available regardless of whether the program was compiled with or without debug options.

Example uses of `truss`

The first example uses a simple program `cdumper.c` with `truss`. The first part involves using the default behavior of `truss`, which follows system calls and the next part is limited to user-level functions.

CODE EXAMPLE 1 **`cdumper.c`**

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

void foo1(int arg1);
void foo2(int arg1);
void foo3(int arg1,double farg1);

main(){
        foo1(1);
}

void foo1(int arg1)
{
        foo2(arg1+16);
}

void foo2(int arg1)
{
        foo3(arg1+6,0.567);
}

void foo3(int arg1,double farg1)
{
        double *pd;
        double  d;

        pd=(double *)arg1;
        *pd=farg1;
}
```

```
% truss cdumper
execve("cdumper", 0xFFBEF33C, 0xFFBEF344)  argc = 1
stat("cdumper", 0xFFBEF088)                     = 0
open("/var/ld/ld.config", O_RDONLY)             Err#2 ENOENT
open("/usr/openwin/lib/libc.so.1", O_RDONLY)    Err#2 ENOENT
open("/usr/lib/libc.so.1", O_RDONLY)            = 3
fstat(3, 0xFFBEEE64)                            = 0
mmap(0x00000000, 8192, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0xFF3B0000
mmap(0x00000000, 778240, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0)
= 0xFF280000
mmap(0xFF336000, 24464, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED, 3, 679936) = 0xFF336000
mmap(0xFF33C000, 6564, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_ANON, -1, 0) = 0xFF33C000
munmap(0xFF326000, 65536)                       = 0
mmap(0x00000000, 8192, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_ANON, -1, 0) = 0xFF3A0000
close(3)                                        = 0
open("/usr/openwin/lib/libdl.so.1", O_RDONLY)   Err#2 ENOENT
open("/usr/lib/libdl.so.1", O_RDONLY)           = 3
fstat(3, 0xFFBEEE64)                            = 0
mmap(0xFF3B0000, 8192, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED, 3, 0) = 0xFF3B0000
close(3)                                        = 0
open("/usr/platform/SUNW,Ultra-60/lib/libc_psr.so.1", O_RDONLY) =
3
fstat(3, 0xFFBEED04)                            = 0
mmap(0x00000000, 8192, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0xFF390000
mmap(0x00000000, 16384, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0xFF380000
close(3)                                        = 0
munmap(0xFF390000, 8192)                        = 0
    Incurred fault #5, FLTACCESS  %pc = 0x000108E8
      siginfo: SIGBUS BUS_ADRALN addr=0x00000017
Received signal #10, SIGBUS [default]
      siginfo: SIGBUS BUS_ADRALN addr=0x00000017
        *** process killed ***
```

```
% truss -t \!all -u a.out cdumper
-> _init(0x0, 0xff336000, 0x10930, 0x0)
<- _init() = 0
-> main(0x1, 0xffbef33c, 0xffbef344, 0x20800)
-> foo1(0x1, 0x0, 0x0, 0x0)
    -> foo2(0x11, 0x0, 0x0, 0x0)
      -> foo3(0x17, 0x3fe224dd, 0x2f1a9fbe, 0x0)
    Incurred fault #5, FLTACCESS  %pc = 0x000108E8
siginfo: SIGBUS BUS_ADRALN addr=0x00000017
    Received signal #10, SIGBUS [default]
      siginfo: SIGBUS BUS_ADRALN addr=0x00000017
        *** process killed ***
```

The following is a simple example of using `truss` to find files being opened by an executable. We will supply the `diff` command with two filenames, the second file being non-existent. In many cases, the error output is not as clear as what the `diff` command clearly states:

```
% diff cdumper.c cdumper.mia
diff: cdumper.mia: No such file or directory
% truss -f -o diff.bad diff cdumper.c cdumper.mia
diff: cdumper.mia: No such file or directory
% egrep "open|stat" diff.bad
1155:   stat("/usr/bin/diff", 0xFFBEF068)               = 0
1155:   open("/var/ld/ld.config", O_RDONLY)           Err#2 ENOENT
1155:   open("/usr/openwin/lib/libc.so.1", O_RDONLY)    Err#2
ENOENT
1155:   open("/usr/lib/libc.so.1", O_RDONLY)            = 3
1155:   fstat(3, 0xFFBEEE44)                           = 0
1155:   open("/usr/openwin/lib/libdl.so.1", O_RDONLY)   Err#2
ENOENT
1155:   open("/usr/lib/libdl.so.1", O_RDONLY)           = 3
1155:   fstat(3, 0xFFBEEE44)                           = 0
1155:   open("/usr/platform/SUNW,Ultra-60/lib/libc_psr.so.1",
O_RDONLY) = 3
1155:   fstat(3, 0xFFBEECE4)                           = 0
1155:   stat("cdumper.c", 0x00028670)                  = 0
1155:   stat("cdumper.mia", 0x000286F8)              Err#2 ENOENT
%
```

Note that instead of using command arguments with `truss` to reduce the calls traced, we get the default output, place it in a file `diff.bad`, and then `grep` for what is wanted. This can be handy to limit the views, but not have to repeatedly run the application. Notice that some of the open "failures" are from the runtime linker tracking down shared object files by retrying with a new path when an `open()` fails.

/usr/proc/bin utilities

Though truss's strength is certainly its ability to dynamically follow processes' system interactions through function calls, taking snapshots of system level information can be used as an advantage. The utilities found in /usr/proc/bin provide a wide variety of system level information about running processes or core files (/usr/proc/bin support for core files first supported in Solaris 8 OE). They are an easy way to access the facilities available through the Solaris OE /proc file system (see proc(4)). Most of the commands take process ids or core file names as options, and then report for each instance.

pstack

This may be the most useful of the proc utilities. This outputs the function call stack of processes or core files. This can be one of the most important first steps in finding where the problem exists. Using pstack on a core file will quickly help narrow the search on what code area or library interface to check. For a process that is believed to be "hung," repeatedly running pstack on it, with the possible addition of truss, can help to determine what is going on, and where the program is. Threaded programs are particularly worthwhile to check with pstack, since often if there is a lock problem the function stack of all the threads together will help indicate the problem's location. In the case of threaded programs with race condition bugs, which are often sensitive to compile options, pstack will usually help to narrow the search.

pfiles

The process' file descriptors are sometimes a strong lead for determining output problems and other issues. Particularly in the case of network daemons and client programs, a listing of the open sockets and files can help in debugging. Unfortunately, the output from pfiles does not list filenames, but instead the inodes and device numbers for file descriptors not connected to sockets. However, even seeing which descriptor numbers and how many are open may help, especially with a program that is having output or connection problems. It may be possible to follow the process initially with truss to find out the filenames for each of these descriptors.

pldd

Most programs link in a large number of shared object libraries. Knowing what these libraries are and where they are being linked from can be critical information. One particular example is programs that link in windowing or graphics libraries. Many sites have multiple versions of the same shared libraries installed, and problems may

only effect some users, depending on their environments. `pldd` may help in finding where programs are pulling their shared libraries from, which can sometimes differ from what `ldd` reports for the executable file because of environment settings.

`pmap`

A process's memory map can be very important for initial debugging purposes. Sometimes a code has reached its stack size or datasize (heap) limit, and will then core dump. Using `pmap`, the core file can be examined to determine the memory used by the process. It can also be used to look at a running process to monitor for memory leaks and usage. If the program dies from a signal such as SEGV or BUS, the user may be able to map the fault address to where it was being used in the program using `pmap`.

`pstop`, `prun`

On a UNIX platform, a user typically learns how to stop and start programs using signals, possibly through terminal input. This, unfortunately, is not as clean as one might prefer in some situations. The preferred action would be to literally stop the process from being scheduled, instead of sending it a signal and incurring the effects of signaling. Certainly threaded program's signalling characteristics can be a bit confusing to the non-expert. By using these utilities, stopping and starting of the process is more in line with what would be desired. In bugs that involve stack corruption or related problems, these utilities can be completely necessary to avoid signals. It may also be useful to `pstop` a program for a short time, and come back and `prun` it back alive at a later time. However be careful using these with applications that have timeouts or time related triggers.

`pflags`

This utility will help in knowing the signal that caused a core dump. It will also tell if the program was 32-bit or 64-bit. The utility also tells the contents of the processor registers for processes that are stopped or core files.

`ptree`

This utility makes it easier to trace down all the processes within an application or shell script. It can also track down the process tree for a specified user id, which can come in handy when debugging a whole system. Though one can get this information from the parent process field in `ps` command output, this utility can be a real time saver in tracking down inter-process relationships.

There are other utilities under `/usr/proc/bin` which were not mentioned. See the references section for man page information on these other utilities.

The ability to use some of the `proc` utilities on core files can help in debugging a critical application failure. Often the code may not have been compiled with debug settings, so a debugger may not be all that helpful on the core files. These utilities can give information on what might have transpired, and direction for further exploration.

Example uses of `/usr/proc/bin` utilities

The following example uses the `cdumper.c` code given previously. Note that specifying debug flags during compilation nor source code is not needed to use these utilities. Be aware that compiling with optimization may move code and possibly inline some functions, which will effect the function stack reported. By looking at the function arguments passed in the source code, note that sometimes arguments have non-zero values, but were not passed in as such. For the `pmap`

output, notice the hexadecimal address output in the first column for each of the libraries and the executable, and compare that with the function addresses reported by `pstack`.

```
% cc -o cdumper cdumper.c
% ./cdumper
Bus error (core dumped)
% pstack core
core 'core' of 1086:    ./cdumper
 000108ec foo3     (17, 3fe224dd, 2f1a9fbe, 0, 0, 0) + 1c
 000108a8 foo2     (11, ff33bfac, 20, ff33bfa0, 21dc0, ff29b65c)
+ 20
 0001085c foo1     (1, 0, ff338588, 5, 21dc0, ff29b21c) + c
 00010824 main     (1, ffbef31c, ffbef324, 20800, 0, 0) + 4
 000107f8 _start   (0, 0, 0, 0, 0, 0) + b8
% pldd core
core 'core' of 1086:    ./cdumper
/usr/lib/libc.so.1
/usr/lib/libdl.so.1
/usr/platform/sun4u/lib/libc_psr.so.1
% pmap core
core 'core' of 1086:    ./cdumper
00010000      8K read/exec         /var/tmp/cdumper
00020000      8K read/write/exec   /var/tmp/cdumper
FF280000    664K read/exec         /usr/lib/libc.so.1
FF336000     24K read/write/exec   /usr/lib/libc.so.1
FF33C000      8K read/write/exec   /usr/lib/libc.so.1
FF380000     16K read/exec         /usr/platform/sun4u/lib/
libc_psr.so.1
FF3A0000      8K read/write/exec
FF3B0000      8K read/exec         /usr/lib/libdl.so.1
FF3C0000    128K read/exec         /usr/lib/ld.so.1
FF3E0000      8K read/write/exec   /usr/lib/ld.so.1
FFBEE000      8K read/write/exec     [ stack ]
 total      888K
%
```

The following example is an attempt to trace down the processes related to an `xterm` window (`pid 720`). The `ptree` command is used to get the pids and stack them according to parent/child relationship.

```
% ptree 720
309   /usr/dt/bin/dtlogin -daemon
  452   /usr/dt/bin/dtlogin -daemon
    650   /bin/ksh /usr/dt/bin/Xsession
     695   /usr/dt/bin/sdt_shell -c unsetenv _ PWD; source /home/
user/.lo
     698  tcsh -c unsetenv _ PWD;    source /home/user/.login;
          709   /usr/dt/bin/dtsession
            720   /usr/openwin/bin/xterm -sb -sl 256 -geom 80x50
              731   tcsh
                1290  rlogin host4
                  1291  rlogin host4
```

/bin/gcore

There are many instances where it would be helpful to get a core dump or a copy of the process's image without the process crashing or exiting. The `/bin/gcore` utility will grab a core image of a running process, without causing the process to exit or sending the process a signal.

An important situation where this utility can be handy is when a process is hung or intermittently causing problems and needs to be reset as soon as possible due to user demand. Use the `gcore` utility to grab a coredump from the process for later debugging,

```
% gcore -o dead_daemon.date process-id
```

Once a coredump is obtained, immediately reset the daemon. A debugger and some of the `proc` utilities can then be used to examine the daemon's core image. Multiple core dumps may be needed to really get a handle of the problem. Either way, the problem might be able to be tracked down from the process function stack. With multiple samples of the failure, odds are increased in finding the cause of it.

/bin/coreadm

There may be cases where managing the filenames of core dumps either at the process level or system wide level can be advantageous. Being able to name both the core dump filename and directory may be used to great effect on heavily used systems and clusters.

A simple example of `coreadm` could be a user changing the default behavior of their core dump files. The defaults of just a current working directory file named `core` leads to overwriting, and thus, lost dumps, when multiple occur; instead, use `coreadm` to setup a specific core dump area for problems:

```
% coreadm -p $HOME/corefiles/core.%n.%f.%p $$
```

This allows core files for each failure, if a set of applications is running, or the same application with different data sets.

A system administrator's desire to stop user program core dumps from clogging the network with NFS traffic could be managed by using `coreadm` to redirect them to local file systems.

The man page for `coreadm` is quite detailed, and the reader is advised to reference it for further information and ideas.

LD_PRELOAD: Runtime Linker Methods

Sometimes it would be advantageous to change the code that an application executes without recompiling the source. In the case of functions that are dynamically loaded, there is a way to do this fairly easily. However, it requires UNIX platform development experience. These techniques are not the first choice to either debug or fix a problem, but they can be very useful or even critical in some cases.

The Solaris OE runtime linking interface allows a user to replace dynamically loaded functions and even augment them. The techniques and concepts are covered in the Solaris OE "*Linker and Libraries Guide.*" The usual method is to use the `LD_PRELOAD` environment variable to force other object files, which can be created by the reader, to be linked into the executable first. This causes the replacement functions to take

the place of the function that would normally be loaded later from a shared object library. A simple example of replacing the system `time()` function which is called from the `date` command is provided:

```
% cat time1.c
#include <sys/types.h>
#include <time.h>

time_t time(time_t *tloc)
{
        if(tloc!=0){
                *tloc=(time_t)0;
        }
        return((time_t)0);
}

% cc -c time1.c
% setenv LD_PRELOAD ./time1.o
% date
Wed Dec 31 19:00:00 EST 1969
% unsetenv LD_PRELOAD
% date
Tue Aug 22 15:54:26 EDT 2000
```

This method does not always have to completely replace a function. It can also be used to augment or slightly change its behavior. The following example shows how to use the fact that many libraries have their functions aliased in a way that `func` is an alias to `_func`. This allows a user to interpose their own `func`, and still call the original routine through `_func`.

```
% cat time2.c
#include <sys/types.h>
#include <time.h>

time_t time(time_t *tloc)
{ time_t  tmptime;
        /* the time symbol is really a weak symbol alias to
           the symbol _time, we'll use this fact to interpose
           our time function but still use the system _time */
        tmptime=_time((time_t *)0);
        tmptime+=3600;
        if(tloc!=0){
                *tloc=tmptime;
        }
        return(tmptime);
}
% cc -c time2.c
% date
Tue Aug 22 16:04:20 EDT 2000
% setenv LD_PRELOAD ./time2.o
% date
Tue Aug 22 17:04:20 EDT 2000
```

In many cases LD_PRELOAD is used to gather more information about a problem. For example one might replace the `open()` system call and force it to print debug information through standard I/O if it fails or output a timestamp. LD_PRELOAD can be used to fix a problem in some cases. If a function is returning an incorrect value or is provided an invalid parameter, this method can be used to fix or avoid the problem.

The reader is strongly cautioned to be careful when using `LD_PRELOAD` with system calls and standard library functions. This should only be done if the reader has a solid grasp of the function's usage and ramifications.

watchmalloc

Some of the more difficult problems to debug are memory corruptions. They can often depend on compile options, dataset sizes, timing, or other processes executing on the system. Worse yet, the failures can occur in functions that are unrelated to the

functions that originally created the problems. In cases where the problem could be memory management or variable initialization, the `libwatchmalloc.so.1` library, part of the Solaris OE, may be helpful.

A strong point of `watchmalloc` is that it can be used with an application without the application being recompiled. The `watchmalloc` routines are invoked by using LD_PRELOAD. A simple example csh usage would be:

```
% setenv LD_PRELOAD libwatchmalloc.so.1
% a.out
```

If `a.out` has un-initialized pointers which are incorrectly assumed to be zero, or is freeing and allocating memory in an improper manner, it will likely fail each time it is run. This may seem of only limited help since it is essentially making matters worse. But memory corruptions can often be intermittent and thereby confusing, so forcing the problem into one that is consistently reproducible can help tremendously.

Further information is available in the `watchmalloc` man page, which discusses advanced usage and limitations.

Symbolic Debuggers

The standard symbolic debuggers provided with the Solaris OE are not for the faint of heart, or those who do not enjoy parsing and converting hexadecimal values. From this article's title, symbolic debuggers may not be within topic. However, they do deserve mention and a window into their potential. They certainly lay outside of many users' and developers' views of a debugger, and thus qualify as "alternate."

`/usr/bin/adb` and `/usr/bin/mdb`

Although the utilities reviewed so far have strengths, they are not without some significant limitations. For example, `truss`, being a dynamic utility, is not applicable when only post-mortem information is available to work with. The `/usr/proc/bin` utilities are good at taking limited snapshots of system level information, but their scope and depth are a major limitation. Getting from the implicated areas of a program to the root cause may require more in-depth debug work.

In most cases, programs are not thoroughly stripped of debug level information. Even if the program is not compiled with debug flags on, there are likely still listings for the program's functions and global variables available for perusal. Thus, `adb`, or, starting with Solaris 8 OE, `mdb`, can be used to look at and manipulate some of the internal information from a program. In some circumstances, the problem could relate to a globally defined variable. Maybe that variable or structure becomes

corrupted and is implicated from the function stack trace. One might be able to use adb on the core file and gather some more information, either to pass onto support organizations, or to use by the developer as a seed for future testing and code review while trying to find the root cause.

Symbolic debuggers are likely the last resort for most situations. However, given some luck and some hard work, the problem may be able to be traced, or even fixed. To use a symbolic debugger on a program effectively requires the user to have a good idea of the applications source code, and access to its header files.

Recommendations for Difficult Debug Sessions

The following recommendations will help in working through difficult debugging sessions; they are also beneficial for simple debug work.

Proceed logically- random debug rarely works

Trying to resolve a problem requires knowing exactly the nature of the problem, and while changing the environment, keeping track of results. By not proceeding in a logical fashion and watching for changes, chances of resolving the problem are greatly reduced, and chances of wasting a lot of time increase. Proceed logically and reduce the problem to the bare facts of its manifestation, and then try to narrow down the cause.

Divide and conquer

In a sense you are at war with the bug, so one of the basic tenets of war should be followed—divide and conquer. This should be applied to all facets of the debugging process—reducing the parameter space of the problem when searching and testing out remedies. Always go back and test out the fixes; check the application with and without the fixes to make sure the problem has really been solved.

Be wary of the multiple problem situation

Cases where multiple problems are effecting the situation are rare, but can be intensely difficult. This can lead to frustration, due to the myriad results from applying possible fixes. Sometimes, the case is such that one problem is unconsciously avoided or worked around, but then shows up again when

debugging another problem. Either way, if applying fixes to a problem only changes its behavior or only reduces the manifestations, then it may be a multiple problem situation. In these cases one has to be particularly careful to change as little as possible when troubleshooting to allow for the most precise interpretations changes in behavior.

Watch for race condition or intermittent bugs

Intermittent problems often take creativity and resourcefulness just to reproduce them in order to start debugging. Unfortunately, these cases can often be misdiagnosed. One can mistakenly assume the problem was fixed due to changing or "fixing" something, only to then find it rearing its ugly head again. Worse yet, it could be a case where the problem fails for one user and not others due to its nature, and the problem is discounted until it strikes again with another user. In these cases, proceed to do anything to gather more data on the problem. Core files are often useful, since dead man forensics can always be done on them to try to find the root cause.

Summary

Applications that are crashing, hanging or displaying anomalous behaviors are a critical issue for any software user. Few people will have the resources and skill set to debug the application directly using a source code debugger. In many cases, source code debugging may not even be an option. This article discussed a variety of options open to a Solaris OE user to narrow down the causes and scope of a application failure, including `truss`, `proc` tools, and special features of the Solaris OE runtime linker. Recommendations were also made on how to proceed with debugging application problems.

Bibliography

Sun Products Documentation: `http://docs.sun.com`

Chris Drake and Kimberley Brown, *"PANIC! UNIX System Crash Dump Analysis Handboo*k," Prentice Hall 1995.

"*Linker and Libraries Guide*," Sun Microsystems, Inc. 2000.

Author's Bio: Chris Duncan

Chris Duncan is currently a Software Engineer in the High End Services group at Sun Microsystems. Chris spends the majority of his time at Sun providing support to Sun's High Performance Computing [HPC] customers. His HPC experience extends more than 10 years, and includes application development, tuning, testing, and supporting HPC software on a variety of architectures and environments.