# Debugging with the PTrace(1) Utility

*Jim Blakey*

## Introduction

Ok, we've all been there. You arrive at a client site, expected to solve strange and wonderful problems, some of which have been lingering unsolved for years. Usually there are no tools to use, no debugger, maybe even no original source code. What do you do?

Well, that's why we get paid the big bucks, right?

In this whitepaper I will discuss two little known tools that will allow you access into the inner workings of processes on unix systems. These tools are the ptrace system calls and the /proc filesystem.

Did you ever wonder how debuggers work? How they gain complete control over the process they are debugging? Well, they all use some combination of ptrace and the /proc filesystem. The following examples will give you the basics to write your own quick and dirty 'debugger' application, specifically targeted to solving the problem at hand.

Ptrace is basically a kernel hook into the task dispatch logic. It provides a mechanism by which you can write a program that can attach to a target process, breakpoint it, single step it, and have access to its entire address space, stack, and registers. With this, you will be able have your program trace execution of a target process, monitor 'watch points', and check for various program conditions, even when the client is too cheap to spring for a license for the debugger package.

The /proc filesystem is a pseudo-filesystem that is maintained by the kernel. It provides a direct interface to many kernel data structures, as well as most process data structures. It also will allow you read (and sometimes write) access to a process virtual address space.

Ptrace and the /proc file system exist on most Unix systems. Most debuggers are based on calls to ptrace. The exception to this is Solaris. Whereas ptrace exists on Solaris, it is poorly implemented and has terrible documentation. For some unknown reason, Sun decided that they would implement debugging through extension of the /proc filesystem. I'll cover both approaches here.

This is not a tutorial. Although the man pages for ptrace are usually poorly written (they are written for debugger writers, who usually know this stuff already), they provide a good reference point. I'll provide some interesting examples of code, and some basic descriptions of how it works. The rest is up to you. Again, that's why we get paid the big bucks. But the hope here is that this will provide you with some tools that will help you gain information to solve those 'unsolveable' problems.

## ptrace(1) And How To Use It

### Example Problem 1: The mysterious infinite loop

Assume we have a client application that goes into an infinite loop. We have no further information, and we have no access to any debuggers. All we know is that the application locks up tight, the system bogs down, and the CPU usage pegs out near 100%. Through the 'ps' command, we can identify which process is ringing up the CPU time, but that's about all the info we have.

What we would really like to do is to be able to stop the process and see exactly where the program counter is, and what its stack context looks like. With a debugger this is trivial. Without one, it is a challenge. Once we have the address we're executing, then we can use a link map (or the 'nm' utility) to find the name of the routine being executed. A little more math, and we get the exact offset into the routine. Finally, if we have source, we can narrow it down to the lines.

The following program uses the ptrace system calls to attach to the supplied target PID. The act of attaching effectively breakpoints the target process. Since we know the target is in an infinite loop, we'll fetch the current Instruction Pointer (IP) and Stack Pointer (SP) and then single step the process. As we single step the process, we'll record each new IP and SP, until we come back to the original point we started from. This will give us one complete iteration of the offending infinite loop.

```
/* ****************************************************************
** pt.c
**
** This program is an example of how to use the ptrace(2) feature
** of unix. ptrace provides a means by which a 'debug' process may
** observe and control the execution of a target process. It provides
** mechanisms to examine and change the target core image, registers
** and flow of execution.
**
** This example will attach to a currently running process and
** put it in single step mode (x86 supports this). From then on,
** each instruction the target executes will cause a breakpoint trap
** in the debugging process (received through the wait(2) call). The
** debugging process will read the target's current instruction and
** stack pointers and write them to stdout
*
** For this example, we're expecting the target process to be in an
** infinite loop. We want to trace exactly one iteration of this
** loop for later analysis.
**
** To run this program, invoke it with the PID of the target process
** as the first argument. Output is to a stdout
**
** LINUX x86 SPECIFIC VERSION. Other Unix systems will be similar
**
** jdblakey@innovative-as.com
**
** ****************************************************************
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
#include <sys/ptrace.h>
#include <sys/reg.h>
#include <sys/user.h>
#include <sys/signal.h>

#define M_OFFSETOF(STRUCT, ELEMENT) \
        (unsigned int) &((STRUCT *)NULL)->ELEMENT;

#define D_LINUXNONUSRCONTEXT 0x40000000

int main (int argc, char *argv[])

{

int Tpid, stat, res;
int signo;
int ip, sp;
int ipoffs, spoffs;
int initialSP = -1;
int initialIP = -1;
struct user u_area;


/*
** This program is started with the PID of the target process.
*/
        if (argv[1] == NULL) {
                printf("Need pid of traced process\n");
                printf("Usage: pt  pid  \n");
                exit(1);
        }
        Tpid = strtoul(argv[1], NULL, 10);
        printf("Tracing pid %d \n",Tpid );
/*
** Get the offset into the user area of the IP and SP registers. We'll
** need this later.
*/
        ipoffs = M_OFFSETOF(struct user, regs.eip);
        spoffs = M_OFFSETOF(struct user, regs.esp);
/*
** Attach to the process. This will cause the target process to become
** the child of this process. The target will be sent a SIGSTOP. call
** wait(2) after this to detect the child state change. We're expecting
** the new child state to be STOPPED
*/
```

```
        printf("Attaching to process %d\n",Tpid);
        if ((ptrace(PTRACE_ATTACH, Tpid, 0, 0)) != 0) {;
                printf("Attach result %d\n",res);
        }
        res = waitpid(Tpid, &stat, WUNTRACED);
        if ((res != Tpid) || !(WIFSTOPPED(stat)) ) {
                printf("Unexpected wait result res %d stat %x\n",res,stat);
                exit(1);
        }
        printf("Wait result stat %x pid %d\n",stat, res);
        stat = 0;
        signo = 0;
/*
** Loop now, tracing the child
*/
        while (1) {
/*
** Ok, now we will continue the child, but set the single step bit in
** the psw. This will cause the child to exeute just one instruction and
** trap us again. The wait(2) catches the trap.
*/
                if ((res = ptrace(PTRACE_SINGLESTEP, Tpid, 0, signo)) < 0) {
                        perror("Ptrace singlestep error");
                        exit(1);
                }
                res = wait(&stat);
/*
** The previous call to wait(2) returned the child's signal number.
** If this is a SIGTRAP, then we set it to zero (this does not get
** passed on to the child when we PTRACE_CONT or PTRACE_SINGLESTEP
** it).  If it is the SIGHUP, then PTRACE_CONT the child and we
** can exit.
*/
                if ((signo = WSTOPSIG(stat)) == SIGTRAP) {
                        signo = 0;
                }
                if ((signo == SIGHUP) || (signo == SIGINT)) {
                        ptrace(PTRACE_CONT, Tpid, 0, signo);
                        printf("Child took a SIGHUP or SIGINT. We are done\n");
                        break;
                }
/*
** Fetch the current IP and SP from the child's user area. Log them.
*/
                ip = ptrace(PTRACE_PEEKUSER, Tpid, ipoffs, 0);
                sp = ptrace(PTRACE_PEEKUSER, Tpid, spoffs, 0);
/*
** Checkto see where we are in the process. Using the ldd(1) utility, I
** dumped the list of shared libraries that were required by this process.
** This showed:
**
**     libc.so.6 => /lib/i686/libc.so.6 (0x40030000)
**     /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
**
** So basically, we can assume that any execuable address pointed to by
** the IP that is *over* 0x40000000 is either in ld.so, libc.so, or in
** some sort of kernel state. We really don't care about these addresses
** so we'll skip 'em. Also, nm(1) showed that all the local symbols
** we would be interested in start in the 0x08000000 range.
*/
                if (ip & D_LINUXNONUSRCONTEXT) {
                        continue;
                }
                if (initialIP == -1) {
                        initialIP = ip;
                        initialSP = sp;
                        printf("---- Starting LOOP IP %x SP %x ---- \n",
                                                initialIP, initialSP);
                } else {
                        if ((ip == initialIP) && (sp == initialSP)) {
                                ptrace(PTRACE_CONT, Tpid, 0, signo);
                                printf("----- LOOP COMPLETE -----\n");
                                break;
                        }
                }
                printf("Stat %x IP %x SP %x  Last signal %d\n",stat, ip, sp,
                                                        signo);
/*
** If we're back to where we started tracing the loop, then exit
*/
```

```
        }
        printf("Debugging complete\n");

        sleep(5);
        return(0);
}
```

The nm(1) utility was run against the offending process's executable image. nm(1) reads symbols from the ELF image and prints out starting addresses of any statically linked area. Dymanically resolved symbols are also printed out, but obviously with no address information. Since all local routines are statically linked, you can see the names of each routine in the process.

Other useful utilities along this line are objdump(1) and readelf(1). The objdump(1) utility provides *much* more useful information than the nm(1) utility, dumping all ELF sections, as well as providing disassembly listings of all code sections. Objdump(1) comes as part of the GCC toolset, but may not be available on all systems. The nm(1) utility is, and that is why I used it in this example.

```
results of the 'nm' command. Note that the target process was *not*
compiled/linked with the -g (debugging) option.


080484e8 T CalcIteration
08048558 T SubFunc1
08049664 ? _DYNAMIC
08049638 ? _GLOBAL_OFFSET_TABLE_
080485e4 R _IO_stdin_used
0804962c ? __CTOR_END__
08049628 ? __CTOR_LIST__
08049634 ? __DTOR_END__
08049630 ? __DTOR_LIST__
08049624 ? __EH_FRAME_BEGIN__

               SNIP (for space... we all know
               what the output from nm looks like)

080485c0 t gcc2_compiled.
080484c0 t gcc2_compiled.
080484b0 t init_dummy
080485b0 t init_dummy
080484c0 T main
0804972c b object.2
0804961c d p.0
         U printf@@GLIBC_2.0
         U sleep@@GLIBC_2.0
```

Ok, a few minor things to note. The *main* function starts at 0x080484c0 and runs for (at most) 0x126c bytes. There are a bunch of obvious library calls, and two suspicously named functions, *CalcIteration*, which starts at 0x080484e8 and runs for 0x6d bytes, and *SubFunc1*, which starts at 0x08048558 and runs for 0x110c bytes.

Now, the output from our little 'debugger' program

```
Tracing pid 2469
Attaching to process 2469
Wait result stat 137f pid 2469
---- Starting LOOP IP 8048568 SP bffff900 ----
Stat 57f IP 8048568 SP bffff900  Last signal 0
Stat 57f IP 804856b SP bffff910  Last signal 0
Stat 57f IP 804856e SP bffff910  Last signal 0
Stat 57f IP 8048571 SP bffff910  Last signal 0
Stat 57f IP 8048573 SP bffff910  Last signal 0
Stat 57f IP 8048575 SP bffff910  Last signal 0
Stat 57f IP 8048577 SP bffff910  Last signal 0
Stat 57f IP 8048578 SP bffff91c  Last signal 0
Stat 57f IP 804852a SP bffff920  Last signal 0
Stat 57f IP 804852d SP bffff930  Last signal 0
Stat 57f IP 804852f SP bffff930  Last signal 0
Stat 57f IP 8048532 SP bffff930  Last signal 0
Stat 57f IP 8048534 SP bffff930  Last signal 0
Stat 57f IP 8048537 SP bffff92c  Last signal 0
```

```
Stat 57f IP 804853a SP bffff928  Last signal 0
Stat 57f IP 804853d SP bffff924  Last signal 0
Stat 57f IP 8048542 SP bffff920  Last signal 0
Stat 57f IP 8048390 SP bffff91c  Last signal 0
Stat 57f IP 8048547 SP bffff920  Last signal 0
Stat 57f IP 804854a SP bffff930  Last signal 0
Stat 57f IP 804854d SP bffff930  Last signal 0
Stat 57f IP 804854f SP bffff930  Last signal 0
Stat 57f IP 804850c SP bffff930  Last signal 0
Stat 57f IP 8048510 SP bffff930  Last signal 0
Stat 57f IP 804851c SP bffff930  Last signal 0
Stat 57f IP 804851f SP bffff928  Last signal 0
Stat 57f IP 8048522 SP bffff924  Last signal 0
Stat 57f IP 8048525 SP bffff920  Last signal 0
Stat 57f IP 8048558 SP bffff91c  Last signal 0
Stat 57f IP 8048559 SP bffff918  Last signal 0
Stat 57f IP 804855b SP bffff918  Last signal 0
Stat 57f IP 804855e SP bffff910  Last signal 0
Stat 57f IP 8048561 SP bffff904  Last signal 0
Stat 57f IP 8048563 SP bffff900  Last signal 0
Stat 57f IP 8048370 SP bffff8fc  Last signal 0
----- LOOP COMPLETE -----
Debugging complete
```

Ok, this should be one complete iteration of our infinite loop. Our loop starts at 0x08048568, which we can calculate to be at SubFunc() + 0x10 bytes. From there, we progress normally for 8 instructions, until we jump to 0x0804852a, which we know to be in CalcIteration() + 0x42 bytes. From there, we progress for 9 more instructions till we hit 0x08048390. nm(1) does not show this, but dumpobj() would have shown that this is part of the dynamic relocation jump table, where a call to printf() is resolved. So we can infer that the 0x08048390 is a printf() statement.

Proceeding after the printf() jump we're still in CalcIteration(). We stay in that for the next 10 instructions, till we hit 0x08048558, the starting address of SubFunc(), which we execute for 5 instructions. The last address, 0x08048370 is again part of the dynamic relocation jump table, which resolves to a call to sleep().

So, what have we found out. Well, without being able to see the source code, and without access to a debugger, we know our infinite loop bounces between the routines CalcIteration() and SubFunc1(), and includes one call to printf() and one call to sleep(). Enough to solve the problem? Maybe, maybe not, but it sure is a lot more information than we had when we started.

**Example Problem 2: The munged memory location**

Consider the problem where we know that some memory address is getting corrupted, but we have no idea when or where it is getting overwritten. We've all at times wished for a way to implement a 'watch point', or a way to monitor a known memory location to see when it changes, and exactly what code we were executing when it changed. Some systems now allow this through debuggers, others don't. But we're assuming that we have no access to debuggers, right?

The following example is a program that implements a watchpoint on a memory location using ptrace(2) (assuming that the target system does not have an intrinsic watchpoint functionality... more on this further down). It will tell you the exact address of the instruction that changed the memory location supplied.

```
/* ****************************************************************
** watchpoint.c
**
** Implements a watchpoint on a supplied memory location. This will
** let you know when this address gets overwritten.
**
** usage:
**
**     watchpoing  PID 0xaddress
**
** LINUX x86 SPECIFIC VERSION. Other Unix systems will be similar
**
** jdblakey@innovative-as.com
**
** ****************************************************************
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
```

```
#include <sys/ptrace.h>
#include <sys/reg.h>
#include <sys/user.h>
#include <sys/signal.h>

#define M_OFFSETOF(STRUCT, ELEMENT) \
        (unsigned int) &((STRUCT *)NULL)->ELEMENT;


int main (int argc, char *argv[])

{

int Tpid, stat, res;
int signo;
int ip, sp;
int ipoffs, spoffs;
struct user u_area;
unsigned int memcontents = 0, startcontents = 0, watchaddr = 0;


/*
** This program is started with the PID of the target process and
** the watched address
*/
        if ((argv[1] == NULL) || (argv[2] == NULL)) {
                printf("Need pid of traced process\n");
                printf("Usage: pt  pid 0xwatchaddress\n");
                exit(1);
        }
        Tpid = strtoul(argv[1], NULL, 10);
        watchaddr = strtoul(argv[2], NULL, 16);
        printf("Tracing pid %d. checking for change to %x \n",Tpid ,watchaddr);
/*
** Get the offset into the user area of the IP and SP registers. We'll
** need this later.
*/
        ipoffs = M_OFFSETOF(struct user, regs.eip);
        spoffs = M_OFFSETOF(struct user, regs.esp);
/*
** Attach to the process. This will cause the target process to become
** the child of this process. The target will be sent a SIGSTOP. call
** wait(2) after this to detect the child state change. We're expecting
** the new child state to be STOPPED
*/
        printf("Attaching to process %d\n",Tpid);
        if ((ptrace(PTRACE_ATTACH, Tpid, 0, 0)) != 0) {;
                printf("Attach result %d\n",res);
        }
        res = waitpid(Tpid, &stat, WUNTRACED);
        if ((res != Tpid) || !(WIFSTOPPED(stat)) ) {
                printf("Unexpected wait result res %d stat %x\n",res,stat);
                exit(1);
        }
        printf("Wait result stat %x pid %d\n",stat, res);
        stat = 0;
        signo = 0;
/*
** Get the starting value at the requested watch location. The PTRACE_PEEKEXT
** option allows you to reach into the tartet process address space, using
** its relocation maps, and read/change values. Nice, huh?
*/
        startcontents = ptrace(PTRACE_PEEKTEXT, Tpid, watchaddr, 0);
        printf("Starting value at %x is %x\n",watchaddr, startcontents);
/*
** Loop now, tracing the child
*/
        while (1) {
/*
** Ok, now we will continue the child, but set the single step bit in
** the psw. This will cause the child to exeute just one instruction and
** trap us again. The wait(2) catches the trap.
*/
                if ((res = ptrace(PTRACE_SINGLESTEP, Tpid, 0, signo)) < 0) {
                        perror("Ptrace singlestep error");
                        exit(1);
                }
                res = wait(&stat);
/*
** The previous call to wait(2) returned the child's signal number.
```

```
** If this is a SIGTRAP, then we set it to zero (this does not get
** passed on to the child when we PTRACE_CONT or PTRACE_SINGLESTEP
** it).  If it is the SIGHUP, then PTRACE_CONT the child and we
** can exit.
*/
                if ((signo = WSTOPSIG(stat)) == SIGTRAP) {
                        signo = 0;
                }
                if ((signo == SIGHUP) || (signo == SIGINT)) {
                        ptrace(PTRACE_CONT, Tpid, 0, signo);
                        printf("Child took a SIGHUP or SIGINT. We are done\n");
                        break;
                }
/*
** get the current value from the watched address and see if it is
** different from the starting value. If so, then get the instruction
** pointer from the target's user area, 'cause this is the instruction
** that changed the value!
*/
                memcontents = ptrace(PTRACE_PEEKTEXT, Tpid, watchaddr, 0);
                if (memcontents != startcontents) {
                        ip = ptrace(PTRACE_PEEKUSER, Tpid, ipoffs, 0);
                        printf("!!!!! Watchpoint address changed !!!!!\n");
                        printf("Instruction that changed it at %x\n",ip);
                        printf("New contents of address %x\n",memcontents);
                        break;
                }
        }
        printf("Debugging complete\n");
        return(0);
}
```

Once this prints out the address of the instruction that changed the watched address, then we can use the same techniques as above to find out more about where the program was. Again, sometimes just knowing which routine we were in when the memory address was corrupted is enough to jumpstart the problem solving.

**Other ptrace(1) Functionality**

The following table lists some of the other useful arguments to the ptrace(1) call. Note that different vendors implement various flavors of these, so it would be best to refer to the man pages on the target system. This list provides a sort of generic set of capabilities.

| | |
|---|---|
| PTRACE_ATTACH | This 'attaches' the parent process to the target PID. In otherwords, the process with the supplied PID is stopped and becomes a child of the parent, allowing the parent access into its process space. The parent will be trapped on all child state changes. *This doesn't work so well on Solaris* |
| PTRACE_TRACEME | This is a call from a target process to tell the parent to trace it. This is usually done after the parent fork(2)s the child and before the child exec(3)s the new image. The parent will then be able to trace the child from the start |
| PTRACE_SINGLESTEP | This places the target child in 'singlestep' mode. Since the target process is a child of the ptrace debugging process, the parent will get a child state changed trap (child changed to STOPPTED) that can be detected with the wait(2) call. |
| PTRACE_SYSCALL | This will trap the parent process for each system call (context change into kernel mode) made by the child. Appendix A describes the strace(2) utility which uses this. |
| PTRACE_CONT | This call continues execution of the STOPPED target process. If the target process stopped on a signal, the parent must deliver the signal on to the child through this call. |
| PTRACE_PEEKTEXT<br>PTRACE_POKETEXT | Allows the program to read or write the word at *addr* in the target's TEXT memory area. |
| PTRACE_PEEKUSER<br>PTRACE_POKEUSER | Read or write a word from the target process's USER structure. This structure holds the registers, stack and text start address, context information, and other process information. See sys/user.h |
| PTRACE_GETREGS<br>PTRACE_SETREGS | Read/write a copy of the target's general purpose registers to/from the supplied location. This will be OS and machine architecture specific. |
| PTRACE_GETFPREGS | Read/write a copy of the target's floating point registers |

| | |
|---|---|
| PTRACE_SETFPREGS | to/from the supplied location. This will be OS and machine architecture specific. |

## The /proc File System

/proc is a pseudo-filesystem that allows us to read (and sometimes write) both kernel and process data structures. Before, if we wanted to do this, we had to tip-toe through /dev/kmem, and know all the various kernel data structures. /proc makes this easy. For example, if I want to know the status of a process with the PID 806, I don't have to read /dev/kmem, or write some shell script to loop ps and use grep/awk to filter the ps output. All I have to do is open /proc/806/stat and read it. As a matter of fact, almost all modern implementations of ps(1) use the /proc feature.

Keep in mind that when you are accessing these data structures, you are not accessing copies of the data, you are reading/writing the actual kernel data structures. The /proc filesystem manager is simply a filesystem driver that translates I/O requests into read/writes of known kernel data structures. The values you read are immediate. This allows for amazing power, but also entails the usual risk of crashing the kernel.

For example, in Linux if I use vi and modify the file /proc/sys/net/ipv4/ip_forward to be a 1 instead of a 0, I've dynamically enabled IP forwarding in the kernel from that point on. If I overwrite /proc/kcore, I've overwritten the kernel. Bad news. For that reason, many of the /proc files are read-only, and access to process specific /proc files is limited to the normal user ownership access rules.

**Kernel Access Through /proc**

The /proc filesystem driver on Linux allows for access to many of the kernel data structures, as well as read/write access to kernel tunable parameters. Other Unix flavors implement this differently. Solaris for example, does not allow access to kernel data structures through /proc.

Appendix B describes some of the kernel data structures available on Linux through the /proc filesystem.

**Individual Process Access Through /proc**

*This works for almost all Unix implementations that offer the /proc filesystem. Each vendor implements the structures differently, but the concepts described here should be valid across platforms.*

Every process on a Unix system has its own directory under the /proc filesystem. The directory name is the process ID. Under that directory are files that give visibility (and, for some files, control) into that process. For example the /proc/*PID*/maps will give information on how and where each segment is mapped for that process. The stat (or ps) file will give information on how the process is running (same as the ps command).

On Linux, these files are formatted in ASCII by the /proc filesystem driver. On most other types of Unix, you read them as binary structures. These structures are usually defined in /usr/include/sys/procfs.h.

The following table lists some common files available for each process. Again, this is OS specific.

| | |
|---|---|
| /proc/**PID**/stat | This file contains information about the current status of the process. This is the file the ps utility uses for its listing. Under Linux, this is an ASCII printout. Under Solaris, this is a binary structure found in sys/procfs.h |
| /proc/**PID**/maps | This file contains entries for each currently mapped memory region, its address offset and size, and the read/write/execute permission on it. |
| /proc/**PID**/cmdline | This is a copy of the command line that started the process. |
| /proc/**PID**/fd | This is a subdirectory containing one entry for each File Descriptor the process has open. The entries in this subdirectory are special 'links' to the actual files opened. |
| /proc/**PID**/mem | This is an access point into the memory space for the target process. On Solaris, this is called /proc/**PID**/as. |

These are just some of the available files in /proc. Each Unix flavor offers more, but these are the common ones.

**An example of using /proc**

The following code example searches the target process's address space for valid segments. When it finds valid segments, it reads in 1k chunks from the target's memory and searches these for the supplied text string. When it finds the string, modifies the first character and writes it back to the target. The next time the target prints the string, the first character will be different.

The /proc/**PID**/mem file can be treated just like any other file. Once opened, you can use lseek() to seek to the target address. The proc driver takes care of mapping that to process address space. Then you can read/write as necessary. This

example works under both Linux and Solaris, with some exceptions noted below).

```
/* ---------------------------------------------------------------
** readmem.c
**
** This program is a quick example of using the /proc filesystem
** to access a process memory space.
**
** This process will scan through all addresses on 1k boundaries
** looking for readable segments by lseek()ing through the /proc/PID/mem
** file. Once it finds a valid segment, it will read buffers and search
** for specific values
**
** On Non-Linux systems, it will then change the first character of the
** buffer to an 'A'
**
** To run:
**
**     readproc  PID  "string to search for "
**
** ---------------------------------------------------------------
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/ptrace.h>
#include <sys/user.h>
#include <sys/signal.h>


#define D_LINUX 1


main(int argc, char *argv[])

{
int Tpid;
int pfd;
char buff[1024];
char *srchstr;
char *eptr;
unsigned int addr;
int  goodaddr;
int goodread;
int srchlen;
int j;

/*
** Get arguments. Since this is an example program, I won't validate them
*/
        if ((argv[1] == NULL) || (argv[2] == NULL)) {
                printf("Need PID and string to search for\n");
                printf("usage: readmem PID string\n");
                exit(1);
        }

        Tpid = strtoul(argv[1], &eptr, 10);
        srchstr = (char *)strdup(argv[2]);
        srchlen = strlen(srchstr);
        printf("readproc: Tracing PID %d for string [%s] len %d\n",Tpid,
                                                srchstr, srchlen);
/*
** In order to read /proc/PID/mem from this process, I have to have already
** stopped it via Ptrace(). (This is not documented anywhere, by the
** way). Anyway, this will leave the process in a STOPPED state. We'll
** start it again in a minute...
*/
        if ((ptrace(PTRACE_ATTACH, Tpid, 0, 0)) != 0) {
                printf("procexa: Attached to process %d\n",Tpid);
        }
/*
** Create the string and open the proc mem file. Note under Linux this
** file is /proc/PID/mem while under Solaris this is /proc/PID/as
```

```
** Also, even though we open this RDWR, Linux will not allow us to write
** to it.
*/
        sprintf(buff,"/proc/%d/as", Tpid);
        printf("procexa:opening [%s]\n",buff);
        if ((pfd = open(buff,O_RDWR)) <= 0) {
                perror("Error opening /proc/PID/mem file");
                exit(2);
        }
/*
** Start at zero, lseek and try to read. increment by 1024 bytes. If
** lseek returns a good status, then the address range is mapped. It
** should be readable. Print out start and end of valid mapped address
** ranges.
*/
        goodaddr = 0; goodread = 0;
        for (addr = 0; addr < (unsigned int)0xf0000000; addr += 1024) {
                if (lseek(pfd, addr, SEEK_SET) != addr) {
                        if (goodaddr == 1) {
                                printf("Address: %x RANGE END\n",addr);
                                goodaddr = 0;
                        }
                        continue;
                } else {
                        if (goodaddr == 0) {
                                printf("Address %x RANGE START\n",addr);
                                goodaddr = 1;
                        }
                }
/*
** Read a 1k buffer and search it for the supplied text string
*/
                if (read(pfd, buff, 1024) <= 0) {
                        if (goodread == 1) {
                                printf("READ address %x RANGE END\n",addr);
                                goodread = 0;
                        }
                        continue;
                } else {
                        if (goodread == 0) {
                                printf("READ address %x RANGE START\n",addr);
                                goodread = 1;
                        }
                        for (j = 0; j < 1024; j++) {
                                if (memcmp(&buff[j], srchstr, srchlen) == 0) {
                                        printf("*****Pattern found %x\n",
                                                                        addr + j);
                                buff[j] = 'A';
                                }
                        }
/*
** If NOT Linux, then write the modified buffer back to the address
** space.
*/
#ifndef D_LINUX
                        lseek(pfd, addr, SEEK_SET);
                        if (write(pfd, buff, 1024) <= 0) {
                                printf("Nope on write\n");
                        }
#endif
                }
        }
        printf("Stopped at address %x\n",addr);
}
```

There are a few things to note about this example. First, although it is not documented anywhere, you must first PTRACE_ATTACH to the target process before you can open the /proc memory image file. This is a security feature. Also, unless you're attaching to yourself (/proc/self/mem), you must be root.

Second, there is a bug in the Linux implementation of the /proc/mem driver's lseek logic. Technically, stack space should be available for reading. Stack space starts at 0xbffffxxx. Note that this has the sign bit set. The lseek() code checks for a negative seek offset, and returns an error, where it should treat the offset as unsigned as it is a valid part of the process address space. The kernel /dev/mem driver does this correctly. This will be fixed in a later release of Linux.

Third, under Linux you can not write to the /proc/**PID**/mem file. The standard distribution kernel has the write code in the /proc/mem driver conditionally compiled out. This was done for security reasons, as a non-suid process could fork and ptrace a suid process, then overwrite its executable memory with evil code. Solaris either handles this differently or does not consider this a problem.

Finally, you may notice that the string seached for is found twice. This is because the dynamic loader (ld.so) mmaps two version of the text/data segments into process space. One is read/execute-only, the other is enabled for write. See the /proc/**PID**/maps (or similar) file to verify this.

## Issues With These Approaches

### Heisenbugs

A 'Heisenbug' is a change in the behavior of a bug through the actions of debugging. Sometimes the act of debugging a process will cause the bug you're searching for to disappear, or new, fun and exciting bugs to appear.

Anytime you introduce a debugger into a process, things change. For simple problems, this is usually not an issue. However for timing related problems, or for debugging real-time processes, using a standard debugger can cause more headaches than it solves.

Debugging is an expensive process. For each breakpoint or single step trap, an interrupt is generated, the child's context is saved, the parent's context is loaded, and the parent is dispatched. This can take a lot of time. Then, with a traditional debugger, the meat behind the keyboard has to manually press the 'next' button. The process is then reversed. In computer time, this is eons.

Now, being able to write a program that automatically single-steps a process, or handles a breakpoint, cuts those eons down to computer- managable times. Further, the program can be tailored in such a way that the initial debugging is very lightweight (monitoring, say, an address using the /proc/**PID**/mem file) until a set of conditions is met. Only then does expensive breakpoint/single step debugging kick in.

So it may be possible using these techniques to avoid some of the normal problems encountered debugging timing critical or real-time problems.

## Summary

The usefulness of these features should be obvious. The examples given show several interesting cases, but by no means is that all we can do. For example, we can attach to several processes at once, and watch their interaction. We can use a trigger in one process to start debugging in another. We can tailor our debugging to wait silently for a certain event (memory change, process output, etc) before attaching and debugging. In effect, we are no longer constrained by the limitations of available debuggers.

---

## Appendix A: Special Bonus: the strace(2) utility

The strace(2) utility is tool written using ptrace(2) calls that allow you to trace and log system calls a target process makes. This is very useful for profiling a process's execution, as well as gaining some insite into what the process is doing. For each system call executed, the name, arguments, and return code are logged to stdout.

Note that under Solaris, the strace(1) utility is a *Streams* utility. To trace process system call execution on Solaris, use truss(1).

The following is a listing of a very simple "Hello World" program under Linux. Note that there is a lot happening here under the covers. I'll discuss that in a minute.

```
[jimb@chaos PTrace]$ strace world
execve("./world", ["world"], [/* 34 vars */]) = 0
uname({sys="Linux", node="chaos", ...}) = 0
brk(0)                                  = 0x8049620
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=95241, ...}) = 0
old_mmap(NULL, 95241, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                = 0
open("/lib/i686/libc.so.6", O_RDONLY)   = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0 \306\1"..., 1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=5772268, ...}) = 0
```

```
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x4002f000
old_mmap(NULL, 1290088, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40030000
mprotect(0x40162000, 36712, PROT_NONE)  = 0
old_mmap(0x40162000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x131000) = 0x40162000
old_mmap(0x40167000, 16232, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40167000
close(3)                                = 0
munmap(0x40017000, 95241)               = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 9), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40017000
write(1, "Hello world\n", 12Hello world
)           = 12
munmap(0x40017000, 4096)                = 0
_exit(12)                               = ?
```

This is not difficult to interpret, given what we know about ptrace and a few other bits of information.

Strace forks the target process and uses the PTRACE_TRACEME call before execing it. Therefore, as we would expect, the first system call is execv(). Brk() is the system call to change a process data segment size. The malloc(3) library call will often result in a brk(2) system call.

All ELF format executables run under a dynamic loader (ld.so) that resolves undefined symbols as needed. With that in mind, the next few system calls are from ld.so on behalf of our world.c task. We see it opening several important files (/etc/ld.so.preload and /etc/ld.so.cache) that tell ld.so where to find library files, then opening /lib/i686/libc.so.6 to resolve printf() and other library calls. The calls to old_mmap() are ld.so mapping segments of libc.so.6 into our process space (around 0x400xx000).

Finally, the one executable line of world.c (printf("Hello World\n");) breaks down to a write() system call, where the string "Hello World" is visable.

Keep in mind when analyzing strace logs that ld.so plays an important part during the entire execution life of a process. You will often see signs of its activity as it dynamically resolves symbols during runtime.

## Appendix B: Interesting /proc system files in Linux

*Note that this is Linux specific. The Solaris implementation allows for access to all processes, but kernel data structures are not exposed.*

There are several interesting files in the /proc filesystem that relate to kernel information. For example, the /proc/interrupts file contains information relating to the IRQ allocations. Note how the filesystem driver takes the raw kernel structure and formats it for us as readable ASCII text.

```
[chaos] cat /proc/interrupts

           CPU0
  0:    3016820          XT-PIC  timer
  1:      53769          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  8:          1          XT-PIC  rtc
  9:     107195          XT-PIC  usb-uhci, Ricoh Co Ltd RL5c475, ymfpci, eth0
 12:     177043          XT-PIC  PS/2 Mouse
 14:      94042          XT-PIC  ide0
NMI:          0
ERR:          0
```

So from this we can see which kernel modules are using which IRQs, and a running count of the number of interrupts received on each IRQ. Remember, that these reflect immediate current values for IRQs.

Two other interesting files are the /proc/ioports and /proc/iomem files. These show which I/O ports are used by which kernel modules, and exactly how the I/O memory is mapped. This is very useful for solving memory and IRQ conflicts when writing PC device drivers.

```
[chaos] cat /proc/ioports

0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0070-007f : rtc
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
```

```
01f0-01f7 : ide0
03c0-03df : vga+
03e8-03ef : serial(auto)
03f6-03f6 : ide0
03f8-03ff : serial(set)
0cf8-0cff : PCI conf1
1040-105f : Intel Corp. 82371AB/EB/MB PIIX4 ACPI
4000-40ff : PCI CardBus #02
4400-44ff : PCI CardBus #02
8000-803f : Intel Corp. 82371AB/EB/MB PIIX4 ACPI
fc38-fc3f : Conexant HSF 56k Data/Fax Modem (Mob WorldW SmartDAA)
fc40-fc7f : Intel Corp. 82557/8/9 [Ethernet Pro 100]
  fc40-fc7f : eepro100
fc8c-fc8f : Yamaha Corporation YMF-744B [DS-1S Audio Controller]
fc90-fc9f : Intel Corp. 82371AB/EB/MB PIIX4 IDE
  fc90-fc97 : ide0
  fc98-fc9f : ide1
fca0-fcbf : Intel Corp. 82371AB/EB/MB PIIX4 USB
  fca0-fcbf : usb-uhci
fcc0-fcff : Yamaha Corporation YMF-744B [DS-1S Audio Controller]


[chaos] cat /proc/iomem

00000000-0009f7ff : System RAM
0009f800-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000f0000-000fffff : System ROM
00100000-07feffff : System RAM
  00100000-00211f08 : Kernel code
  00211f09-0028006b : Kernel data
07ff0000-07fff7ff : ACPI Tables
07fff800-07ffffff : ACPI Non-volatile Storage
10000000-10000fff : Ricoh Co Ltd RL5c475
10001000-100013ff : Sony Corporation Memory Stick Controller
10400000-107fffff : PCI CardBus #02
10800000-10bfffff : PCI CardBus #02
40000000-40ffffff : Intel Corp. 440BX/ZX/DX - 82443BX/ZX/DX Host bridge
a0000000-a0000fff : card services
fc000000-fdffffff : PCI Bus #01
  fc000000-fdffffff : Neomagic Corporation NM2380 [MagicMedia 256XL+]
fe400000-febfffff : PCI Bus #01
  fe400000-fe7fffff : Neomagic Corporation NM2380 [MagicMedia 256XL+]
  feb00000-febfffff : Neomagic Corporation NM2380 [MagicMedia 256XL+]
fec00000-fecfffff : Intel Corp. 82557/8/9 [Ethernet Pro 100]
fede0000-fedeffff : Conexant HSF 56k Data/Fax Modem (Mob WorldW SmartDAA)
fedf6000-fedf6fff : Intel Corp. 82557/8/9 [Ethernet Pro 100]
  fedf6000-fedf6fff : eepro100
fedf7000-fedf77ff : Sony Corporation CXD3222 i.LINK Controller
fedf7c00-fedf7dff : Sony Corporation CXD3222 i.LINK Controller
fedf8000-fedfffff : Yamaha Corporation YMF-744B [DS-1S Audio Controller]
  fedf8000-fedfffff : ymfpci
fff80000-ffffffff : reserved
```

Finally, as an example of the flexability of this implementation, follow out some of the directory structure under /proc/sys. For example, look in the /proc/sys/net/ipv4 directory. This contains many files, each describing tunable IP options. You can dynamically modify many of these to change the behavior of the system. These changes take effect immediately.

## Appendix C: Solaris Implementations of /proc Debugging

Ptrace exists on Solaris, but is poorly implemented. Several important features do not work as expected, and the ptrace call itself is very poorly documented. Sun has choosen instead to implement debugging through extending the /proc filesystem. Solaris provides several new files, including /proc/**PID**/ctl and /proc/**PID**/lwpctl, which are writable control files that provide a rich set of process control and debug features.

---

This article and all programs and examples contained within are copyright © 2010 Jim Blakey, St Thomas, USVI. All programs and examples are for education and entertainment purposes only. There is no warranty, expressed or implied, on accuracy or usefulness of these examples.

Programs and examples are released under the terms of the GPL. Text may be used with permission of the author.

jdblakey@innovative-as.com