Native Kernel Debugging with Acid

Tad Hunt
tad@plan9.bell-labs.com
Lucent Technologies Inc
(Revised 22 May 2000 by Vita Nuova)

Introduction

This tutorial provides an introduction to the Acid debugger. It assumes that you are familiar with the features of a typical source-level debugger. The Acid debugger is built round a command language with a syntax similar to C. This tutorial is not an introduction to Acid as a whole, but offers a brief tour of the basic built in and standard library functions, especially those needed for debugging native Inferno kernels on a target board.

Acid was originally developed by Phil Winterbottom to help debug multi-threaded programs in the concurrent language Alef, and provide more sophisticated debugging for C programs. In the paper *Acid: A Debugger Built From a Language*, Winterbottom discusses Acids design, including some worked examples of unusual applications of Acid to find memory leaks and assist code coverage analysis. Following that is the *Acid Reference Manual*, also by Phil Winterbottom, which gives a more precise specification of the Acid debugging language and its libraries.

Preliminaries -- the environment

Acid runs under the host operating system used for cross-development, in the same way as the Inferno compilers. Before running either compilers or Acid, the following environment variables must be set appropriately:

ROOT	the directory in which Inferno lives (eg, /usr/inferno).
SYSHOST	*host* operating system type: Nt , Solaris , Plan9 , Linux or FreeBSD
OBJTYPE	*host* machine s architecture type: 386 , sparc , mips , or powerpc

They might be set by a login shell profile (eg, Unix .profile , or Plan 9 lib/profile). Also ensure that the directory

```
$ROOT/$SYSHOST/$OBJTYPE/bin
```

is on your search path. For example, on a Solaris sparc, one might use:

```
ROOT=inferno_root
SYSHOST=Solaris
OBJTYPE=sparc
ACIDLIB=$ROOT/lib/acid
PATH=$ROOT/$SYSHOST/$OBJTYPE/bin:$PATH
export ROOT ACIDLIB PATH OBJTYPE SYSHOST
```

where *inferno_root* is the directory in which Inferno lives (eg, /usr/inferno).

An Example Program

The first example is not kernel code, but a small program that will be compiled but not run, to demonstrate basic Acid commands for source and object file inspection. The code is shown below:

```
int
factorial(int n)
{
    if (n == 1)
        return 1;
    return n * factorial(n-1);
}

int f;
void
main(void)
{
    f = factorial(5);
}

void
_main(void)
{
    main();
}
```

Compiling and Linking

The first step is to create an executable. The example shows the process for creating ARM executables. Substitute the appropriate compiler and linker for other cpu types.

```
% 5c factorial.c
% 5l -o factorial factorial.5
% ls
factorial
factorial.5
factorial.c
```

Starting Acid

Even without the target machine on which to run the program, many Acid features are available. The following command starts debugging the factorial executable. Note that, upon startup, Acid will attempt to load some libaries from the directory specified in the ACIDLIB environment variable (defaults to /usr/inferno/lib/acid). It will also attempt to load the file $HOME/lib/acid, in which you can place commands to be executed during startup.

```
% acid factorial
factorial:Arm plan 9 executable

$ROOT/lib/acid/port
$ROOT/lib/acid/arm
acid:
```

Exploring the Executable

To find out what symbols are in the program:

```
acid: symbols("")
etext     T    0x00001068
f    D    0x00002000
setR12    D    0x00002ffc
end  B    0x00002008
bdata     D    0x00002000
edata     D    0x00002008
factorial      T    0x00001020
main      T    0x00001048
_main     T    0x0000105c
acid:
```

The output from the symbols() function is similar to the output from the *nm* (10.1) command. The first column is the symbol name, the second column gives the section the symbol is in, and the third column is the address of the symbol.

There is also a symbols global variable. Variables and functions can have the same names. It holds the list of symbol information that the symbols function uses to generate the table:

```
acid: symbols
{{"etext", T, 0x00001068}, {"f", D, 0x00002000}, {"setR12", D, 0x00002ffc},
 {"end", B, 0x00002008}, {"bdata", D, 0x00002000}, {"edata", D, 0x00002008},
 {"factorial", T, 0x00001020}, {"main", T, 0x00001048}, {"_main", T, 0x00001
05c}}
acid:
```

In large programs, finding the symbol you are interested in from a list that may be thousands of lines long would be difficult. The string argument of symbols() is a regular expression against which to match symbols. All symbols that contain the pattern will be displayed. For example:

```
acid: symbols("main")
main      T    0x00001048
_main     T    0x0000105c
acid: symbols("^main")
main      T    0x00001048
acid:
```

The symbols function is written in the *acid* command language and lives in the port library ($ACIDLIB/port).

```
defn symbols(pattern)
{
     local l, s;

     l = symbols;
     while l do {
          s = head l;
          if regexp(pattern, s[0]) then
               print(s[0], "", s[1], "", s[2], "0);
          l = tail l;
     }
}
```

Acid retrieves the list of symbols from the executable and turns each one into a global variable whose value is the address of the symbol. If the symbol clashes with a builtin name or keyword or a previously defined function, enough $ characters are prepended to the name to make it unique. The list of such renamings is printed at startup.

Most acid functions operate on addresses. For example, to view the source code for a given address, use the src function:

```
acid: src(main)
/usr/jrf/factorial.c:10
 5       return n * factorial(n-1);
 6   }
 7
 8   int f;
 9   void
>10  main(void)
 11  {
 12      f = factorial(5);
 13  }
 14
 15  void
```

The src(*addr*) function displays a section of source code, with the line containing the address passed as an argument in the middle of the display. To print the assembly code beginning at a given address, use the asm() function.

```
acid: asm(factorial)
factorial   0x00001020      MOVW.W   R14,#-0x8(R13)
factorial+0x4    0x00001024      CMP.S    $#0x1,R0
factorial+0x8    0x00001028      MOVW.EQ  $#0x1,R0
factorial+0xc    0x0000102c      RET.EQ.P      #0x8(R13)
factorial+0x10    0x00001030     MOVW     R0,n+0(FP)
factorial+0x14    0x00001034     SUB  $#0x1,R0,R0
factorial+0x18    0x00001038     BL   factorial
factorial+0x1c    0x0000103c     MOVW     n+0(FP),R2
factorial+0x20    0x00001040     MUL  R2,R0,R0
factorial+0x24    0x00001044     RET.P    #0x8(R13)
main  0x00001048   MOVW.W   R14,#-0x8(R13)
acid:
```

The output contains the symbolic address (symbol name+offset, where symbol name is the name of the enclosing function) followed by the absolute address, followed by the disassembled code. The asm(*addr*) function prints the assembly beginning at *addr* and ending after either 30 lines have been printed, or the end of the function has been reached. The casm() function continues the assembly listing from where it left off, even past the end of the function and into the next one.

```
acid: casm()
main+0x4   0x0000104c   MOVW     $#0x5,R0
main+0x8   0x00001050   BL   factorial
main+0xc   0x00001054   MOVW     R0,$f-SB(SB)
main+0x10   0x00001058      RET.P    #0x8(R13)
_main  0x0000105c       MOVW.W   R14,#-0x4(R13)
acid:
```

All the functions presented so far are written in the acid command language. To see the source of a comand written in the acid command language, use the builtin command whatis [*name*]. It prints the definition of the optional argument *name*. If *name* is an Acid builtin, whatis prints builtin function.

```
acid: whatis casm
defn casm() {
        asm(lasmaddr);
}
acid:
acid: whatis atof
builtin function
acid:
```

If *name* is a variable, it prints the type of variable, and for the integer type, gives the format code used to print the value:

```
acid: whatis pid
integer variable format D
acid:
```

With no arguments, whatis lists all available functions:

```
acid: whatis
Bsrc          bpmask      follow      new        sh
_bpconddel    bpneq       func        newproc    source
_bpcondset    bpor        gpr         next       spr
_stk          bpprint     include     notestk    spsrch
access        bppush      interpret   params     src
acidinit      bpset       itoa        pcfile     start
addsrcdir     bptab       kill        pcline     startstop
asm           casm        kstk        pfl        status
atof          cont        labstk      print      stk
atoi          debug       line        printto    stmnt
bpaddr        dump        linkreg     procs      stop
bpand         error       lkstk       rc         stopped
bpconddel     file        locals      readfile   strace
bpcondset     filepc      lstk        reason     symbols
bpdel         findsrc     map         regexp     waitstop
bpderef       fmt         match       regs
bpeq          fnbound     mem         setproc
acid:
```

The Bsrc(*addr*) function brings up an editor on the line containing *addr*. It simply invokes a shell script named B that takes two arguments, -*line* and *file*. The shell script invokes $EDITOR + *line file*. If unset, EDITOR defaults to *vi*. The shell script, or the Bsrc function can be easily rewritten to work with your favorite editor.

Entering a symbol name by itself will print the address of the symbol. Prefixing the symbol name with a * will print the value at the address in the variable. Continuing to use our factorial example:

```
acid: f
0x00002000
acid: *f
0x00000000
acid:
```

Remote Debugging

Now that you have a basic understanding of how to explore the executable, it is time to examine a real remote debugging session.

We'll use the SA1100 keyboard driver as an example. Examining the kernel configuration file, you'll see the following:

```
dev
        keyboard
link    driver/keyboard    port
        scanfujn860        kbd.h  keycodes.h
link    ./../driver        plat
        kbdfujitsu         ./../common/ssp.h    \
                           /driver/keyboard/kbd.h     \
                           /driver/keyboard/keycodes.h
port
        const char *defaultkeyboard   = "fujitsu";
        const char *defaultkeytable   = "scanfujn860";
        int debugkeys  = 1;      /* 1 = enabled,  0 = disabled  */
```

This describes the pieces of the keyboard driver which are linked into the kernel. The source code lives in two places, $ROOT/os/driver/keyboard, and $ROOT/os/plat/sa1100/driver.

The next step is to build a kernel. Use the *mk* target acid to ensure that the Acid symbolic debugging data is produced. For example:

```
% mk 'CONF=sword'  acid  isword.p9.gz
```

This creates the Acid file isword.acid, containing Acid declarations describing kernel structures, the kernel executable isword.p9; and finally *gzip* s a copy of the kernel in isword.p9.gz to load onto the device. Next, copy the gzipped image onto the device and then boot it. Follow the directions found elsewhere for details of this process.

From a shell prompt on the target device, start the remote debugger by writing the letter r (for run) to #b/dbgctl. Next, start Acid in remote debug mode, specifying the serial port it is connected to with the -R option. $CONF is the name of the configuration file used, for example sword.

```
% acid  -R /dev/cua/b   -l i$CONF.acid   i$CONF
isword:Arm  plan 9 executable
$ROOT/lib/acid/port
i$CONF.acid
$ROOT/lib/acid/arm
/usr/jrf/lib/acid
acid:
```

You are now debugging the kernel that is running on the target device. All of the previously listed commands will work as described before; in addition, there are many more commands available.

Kernel Process Listing

To get a list of kernel processes, use the ps() function:

```
acid: ps()
PID     PC          PRI     STATE     NAME
1       0x00054684  5       Queueing           interp
2       0x00000000  1       Wakeme    consdbg
3       0x00000000  5       Wakeme    tcpack
4       0x00000000  5       Wakeme    Fs.sync
5       0x00000000  4       Wakeme    touchscreen
6       0x00054684  5       Queueing           dis
7       0x00059788  5       Wakeme    dis
8       0x00054684  5       Queueing           dis
9       0x00054684  5       Queueing           dis
10      0x00054684  5       Wakeme    dis
11      0x0004c26c  1       Running   dbg
acid:
```

The PC column shows the address the process was executing at when the ps command retrieved statistics on it. The PRI column lists process priorities. The smaller the number the higher the process priority. Notice that the kernel process (kproc) running the debugger is the highest priority process in the system. The only process you will ever see in the Running state while executing the ps command will be the debugger, since it is gathering information about the other processes.

Breakpoints

Breakpoints in Inferno, unlike most traditional kernel debuggers, are conditional breakpoints. There are minimally two conditions which must be met. These conditions are address and process id. A breakpoint will only be taken when execution for a specific kernel process reaches the specified address. The user can create additional conditions that are evaluated if the address and process id match. If evaluation of these conditions result in a nonzero value, the breakpoint is taken, otherwise it is ignored, and execution continues.

Again, the best way to proceed is with an example:

```
acid: setproc(7)
```

The setproc(*pid*) function selects a kproc to which later commands will be applied; the one with process ID (*pid*) in this case.

```
acid: bpset(keyboardread)
Waiting...
7: stopped       flush8to4+0x18c    MOVW     (R3<<#4),R3
```

After selecting a kproc, we set a breakpoint at the address referred to by the keyboardread symbol. As described before, the value of a global variable created from a symbol in the executable is the address of the symbol. In this case the address is the first instruction in the keyboardread() function. Notice that setting a breakpoint stops the kproc from executing. A bit later, we ll see how to get it to continue execution.

Next, display the list of breakpoints using bptab() :

```
acid: bptab()
ID      PID     ADDR                          CONDITIONS
0       7       keyboardread    0x0003c804    { }
```

The first column is a unique number that identifies the breakpoint. The second column is the process ID in which the breakpoint will be taken. The third and fourth columns are the address of the breakpoint, first in symbolic form, then in numeric form. Finally, the last column is a list of conditions to evaluate whenever the kproc specified in the PID column hits the the address specified in the ADDR column. When they match, the list of conditions is evaluated. If the result is nonzero, the breakpoint is taken. Since we used the simplified breakpoint creation function, bpset(), there are no additional conditions. Later on, we'll see how to set conditional breakpoints.

Start the selected kproc executing again, and wait for it to hit the breakpoint.

```
acid: cont()
```

The cont() function will not return until a breakpoint has been hit, and there is no way to interrupt it. This means you should only set breakpoints that will be hit, otherwise you'll have to reboot the target device and restart your debugging session.

To continue our example, repeatedly hit new line (return, enter) on the keyboard on the target device, until the breakpoint occurs:

```
break 0: pid 7: stopped keyboardread      SUB      $#0xa4,R13,R13
acid:
```

This message, followed by the interactive prompt returning, tells you that a breakpoint was hit. It gives the breakpoint id, the kernel process id, then the symbolic address at which execution halted, followed by the disassembly of the instruction at that address.

The kstk() function prints a kernel stack trace, beginning with the current frame, all the way back to the call that started the kproc. For each function, it gives the name name, arguments, source file, and line number, followed by the symbolic address, source file, and line number of the caller.

```
acid: kstk()
At pc:247812:keyboardread /usr/inferno/os/driver/keyboard/devkey
board.c:350
keyboardread(offset=0x0000009d,buf=0x001267f8,n=0x00000001) /usr
/inferno/os/driver/keyboard/devkeyboard.c:350
        called from kchanio+0x9c /usr/inferno/os/port/sysfile.c:
75
kchanio(buf=0x001267f8,n=0x00000001,mode=0x00000000) /usr/infern
o/os/port/sysfile.c:64
        called from consread+0x144 /usr/inferno/os/driver/port/d
evcons
consread(offset=0x0000009d,buf=0x0043d4fc,n=0x00000400,c=0x0044e
c38) /
usr/inferno/os/driver/port/devcons.c:357
        called from kread+0x164 /usr/inferno/os/port/sysfile.c:2
97
kread(fd=0x00000006,n=0x00000400,va=0x0043d4fc) /usr/inferno/os/
port/sysfile.c:272
        called from Sys_read+0x84 /usr/inferno/os/port/inferno.c
:244
Sys_read() /usr/inferno/os/port/inferno.c:229
        called from mcall+0x98 /usr/inferno/interp/xec.c:590
mcall() /usr/inferno/interp/xec.c:569
        called from xec+0x128 /usr/inferno/interp/xec.c:1098
xec(p=0x0044edd8) /usr/inferno/interp/xec.c:1077
        called from vmachine+0xbc /usr/inferno/os/port/dis.c:706
vmachine() /usr/inferno/os/port/dis.c:677
        called from _main+0x50 /usr/inferno/os/plat/sa1100/infern
o/main.c:237
acid:
```

There is another kernel stack dump function, lkstk(), which shows the same information as kstk() plus the names and values of local variables. Notice that in addition to the called from information, each local variable and its value is listed on a line by itself.

```
acid: lkstk()
At pc:247812:keyboardread /usr/inferno/os/driver/keyboard/devkeyboard.
c:350
keyboardread(offset=0x00000018,buf=0x001267f9,n=0x00000001) /usr/inferno
/os/driver/keyboard/devkeyboard.c:350
        called from kchanio+0x9c /usr/inferno/os/port/sysfile.c:75
        tmp=0x00000000
kchanio(buf=0x001267f9,n=0x00000001,mode=0x00000000) /usr/inferno/os/por
t/sysfile.c:64
        called from consread+0x144 /usr/inferno/os/driver/port/devcons
        c=0x0045a858
        r=0x00000001
consread(offset=0x00000015,buf=0x0043d4fc,n=0x00000400,c=0x0044ec38) /us
r/inferno/os/driver/port/devcons.c:357
        called from kread+0x164 /usr/inferno/os/port/sysfile.c:297
        r=0x00000001
        ch=0x0000006c
        eol=0x00000000
        i=0x00000000
        mt=0x60000053
        tmp=0x0007317c
        l=0x0044ec38
        p=0x00049754
kread(fd=0x00000006,n=0x00000400,va=0x0043d4fc) /usr/inferno/os/port/sys
file.c:272
        called from Sys_read+0x84 /usr/inferno/os/port/inferno.c:244
        c=0x0044ec38
        dir=0x00000000
Sys_read() /usr/inferno/os/port/inferno.c:229
        called from mcall+0x98 /usr/inferno/interp/xec.c:590
        f=0x0044eff0
        n=0x00000400
mcall() /usr/inferno/interp/xec.c:569
        called from xec+0x128 /usr/inferno/interp/xec.c:1098
        ml=0x0043d92c
        f=0x0044eff0
xec(p=0x0044edd8) /usr/inferno/interp/xec.c:1077
        called from vmachine+0xbc /usr/inferno/os/port/dis.c:706
vmachine() /usr/inferno/os/port/dis.c:677
        called from _main+0x50 /usr/inferno/os/plat/sa1100/inferno/main.
c:237
        r=0x0044edd8
        o=0x0044ee50
```

The step() function allows the currently selected process to execute a single instruction, and then stop.

```
acid: step()
break 1: pid 7: stopped keyboardread+0x4     MOVW  R14,#0x0(R13)
acid:
```

The bpdel (*id*) command deletes the breakpoint identified by *id*:

```
acid: bpdel(0)
```

The start() command places the kproc back into the state it was in when it was stopped.

```
acid: start(7)
acid:
```

Now lets look at how to set conditional breakpoints.

```
acid: bpcondset(7, keyboardread, {bppush(_startup), bpderef()})
Waiting...
7: stopped          sched+0x20          MOVW      #0xffffff70(R12),R6
acid: bptab()
ID        PID       ADDR                          CONDITIONS
0         7         keyboardread 0x0003c804 {
                                                  {"p", 0x00008020}
                                                  {"*", 0x00000000} }
acid: *_startup = 0
acid: cont()
```

Conditional breakpoints are set with bpcondset() bptab() function shows the list of conditions to apply. The list is a bit confusing to read, but the "p" means push and the "*" means *dereference*.

No matter how much you type on the keyboard, this particular breakpoint will never be taken. That's because before continuing, we set the value at the address _startup to zero, so whenever execution reaches keyboardread in kproc number 7, it pushes the address _startup, then pops it and pushes the word at that address. Since the top of the stack is zero, the breakpoint is ignored.

This contrived example may not be all that useful, but you can use a similar method in your driver to examine some state before making the decision to take the breakpoint.

Examining Registers

There are three commands to dump registers: gpr(), spr() and regs(). The gpr() function dumps the general purpose registers, spr() dumps special purpose registers (such as the PC and LINK registers), and regs() dumps both:

```
acid: regs()
PC        0x0004a3b0 sched+0x20       /home/tad/inf2.1/os/port/proc.c:82
LINK      0x0004b8e8 kchanio+0xa4       /home/tad/inf2.1/os/port/sysfile.c:75
SP        0x00453c4c
R0        0x00458798 R1       0x000fdf9c R2       0x0003c804 R3       0x00000000
R4        0xffffffff R5       0x00000001 R6       0x00458798 R7       0x00000001
R8        0x001267f8 R9       0x00000000 R10      0x0044ee50 R11      0x00029f9c
R12       0x000fc854
acid:
```

Complex Types

When reading in the symbol table, Acid treats all of the symbols in the executable as pointers to integers. This is fine for global integer variables, but it makes examining more complex types difficult. Luckily there is a solution. Acid allows you to create a description for more complex types, and a function which will automatically be called for these complex types. In fact, the compiler can automatically generate the acid code to describe these complex types. For example, if we wanted to print out the devtab structure for the keyboard driver, we can just give it its name:

```
acid: whatis keyboarddevtab
integer variable format a complex Dev
acid: keyboarddevtab
        dc      107
        name    0x0010e0ea
        reset   0x0003c3fc
        init    0x0003c438
        attach  0x0003c5dc
        clone   0x000480d0
        walk    0x0003c600
        stat    0x0003c640
        open    0x0003c680
        create  0x0004881c
        close   0x0003c768
        read    0x0003c804
        bread   0x0004883c
        write   0x0003c968
        bwrite  0x00048900
        remove  0x00048978
        wstat   0x00048998
acid:
```

Acid knows the `keyboarddevtab` variable is of type `Dev`, and it prints it by invoking the function `Dev(keyboarddevtab)`.

```
acid: whatis Dev
complex Dev {
        'D' 0 dc;
        'X' 4 name;
        'X' 8 reset;
        'X' 12 init;
        'X' 16 attach;
        'X' 20 clone;
        'X' 24 walk;
        'X' 28 stat;
        'X' 32 open;
        'X' 36 create;
        'X' 40 close;
        'X' 44 read;
        'X' 48 bread;
        'X' 52 write;
        'X' 56 bwrite;
        'X' 60 remove;
        'X' 64 wstat;
};
```

```
defn Dev(addr) {
        complex Dev addr;
        print("\tdct",addr.dc,"\n");
        print("\tnamet",addr.nameX,"\n");
        print("\tresett",addr.resetX,"\n");
        print("\tinitt",addr.initX,"\n");
        print("\tattacht",addr.attachX,"\n");
        print("\tclonet",addr.cloneX,"\n");
        print("\twalkt",addr.walkX,"\n");
        print("\tstatt",addr.statX,"\n");
        print("\topent",addr.openX,"\n");
        print("\tcreatet",addr.createX,"\n");
        print("\tcloset",addr.closeX,"\n");
        print("\treadt",addr.readX,"\n");
        print("\tbreadt",addr.breadX,"\n");
        print("\twritet",addr.writeX,"\n");
        print("\tbwritet",addr.bwriteX,"\n");
        print("\tremovet",addr.removeX,"\n");
        print("\twstatt",addr.wstatX,"\n");
}
```

Notice the complex type definition and the function to print the type both have the same name. If we know that an address is the address of a complex type, even though acid may not (say we re storing multiple types of data in a void pointer), we can print the complex type by calling the type printing function ourselves.

```
acid: print(fmt(keyboarddevtab,      'X'))
0x00106d50
acid: Dev(0x00106d50)
        dc       107
        name     0x0010e0ea
        reset    0x0003c3fc
        init     0x0003c438
        attach   0x0003c5dc
        clone    0x000480d0
        walk     0x0003c600
        stat     0x0003c640
        open     0x0003c680
        create   0x0004881c
        close    0x0003c768
        read     0x0003c804
        bread    0x0004883c
        write    0x0003c968
        bwrite   0x00048900
        remove   0x00048978
        wstat    0x00048998
acid:
```

Conclusion

This introduction to using Acid for remote debugging Inferno kernels should be enough to get you started. As a tutorial, it only describes how to use some of the features of the debugger, and does not attempt to describe how to do advanced debugging such as writing your own functions, or modifying existing ones. Exploring the source, setting breakpoints, single stepping through code, and examining the contents of variables are the usual uses of a debugger. This tutorial gives examples of all of these.

For a more in depth discussion of the acid command language, and how to write your own acid functions, see the manual page *acid* (10.1) and Phil Winterbottom's papers on the Acid Debugger, reprinted in this volume.

Appendix

There are two important differences between Acid described in the accompanying paper, and Acid as distributed with Inferno for use in kernel debugging.

Connecting Acid to the remote Inferno kernel

A remote Plan 9 kernel can be debugged in the same way as a Plan 9 user process, using the file server *rdbfs* (4). It is a user-level file server on Plan 9 that uses a special debugging protocol on a serial connection to the remote kernel, but on the Plan 9 side serves a file system interface like that of *proc* (3), for use by Acid. Acid therefore does not need any special code to access the remote kernel's memory, or exert control over it.

Inferno's version of Acid currently runs under the host operating systems, which do not support such a mechanism (except for Plan 9). Instead, Acid itself provides a special debugging protocol, with (host) platform-specific interface code to access a serial port. This might well be addressed in future by implementing the native kernel debugger in Limbo.

Handling of breakpoints

The following functions are provided by the Acid library $ROOT/lib/acid/$OBJTYPE for use in native kernel debugging. In several cases they change the behavior described in the Acid manual. The functions are:

```
id = bpset(addr)
id = bpcondset(pid, addr, list)
bppush(val)
bpderef()
bpmask()
bpeq()
bpneq()
bpand()
bpor()
bptab()
addr = bpaddr(id)
bpdel(id)
bpconddel(id)
```

With traditional breakpoints, when a program reaches an address at which a breakpoint is set, execution is halted, and the debugger is notified. In applications programming, this type of breakpoint is sufficient because communicating the break in execution to the debugger is handled by the operating system. The traditional method of handling breakpoints breaks down when program being debugged is the kernel. A breakpoint cannot entirely suspend the execution of the kernel because there is no other program that can handle the communication to the debugger.

Some operating systems solve this problem by including a mini operating system, a self-contained program within the kernel that has its own code to handle the hardware used to communicate with the remote debugger or user. There are many problems with this mechanism. First, the debugger code that lives inside the kernel must duplicate a lot of code contained elsewhere in the kernel. This makes the kernel much bigger, and can increase maintenance costs. Typically this type of debug support treats the kernel as having a single thread of control, so a breakpoint stops everything while the user decides what to do about it. The only places in the kernel breakpoints cannot be set are in the debugger itself, and in the code that handles notifying the debugger of the breakpoint.

The Inferno kernel takes a different approach. The remote debug support is provided by a device driver that makes use of kernel services. Communication with the remote debugger is handled by a kernel process dedicated entirely to that task. All breakpoints can be considered to be minimally conditional on two values. First, the address to take the break at, and second, the kernel process to take the break in. This method allows the kernel debugger to be implemented as a regular Inferno device driver. The device driver can make use of all the APIs available to device drivers; it does not need to be self contained. Additionally, conditional breakpoints can be set anywhere in the kernel, with two exceptions. As with traditional debugger implementations, breakpoints can not be set in the code that handles notifying the debugger of the breakpoint. Unlike traditional implementations, the code that handles the execution and evaluation of the conditions applied to the breakpoint is the only other place breakpoint cannot be set. Since both of these parts of the kernel code are self contained, the user can set breakpoints in any other kernel routines. For example, the user could set a breakpoint in kread(), for a given kernel process, but the debugger can still call kread() itself.

Use of conditional breakpoints can help make the debugging process more efficient. If there is a bug that occurs in the Nth iteration of a loop, with unconditional breakpoints, user intervention is required N-1 times before reaching the state the bug occurs in. Conditional breakpoints give the user the ability to automatically check the value of N, and only take the breakpoint when it reaches the critical value.

The following changed and additional functions in the Acid library provide access to this extended breakpoint support:

Setting Breakpoints

integer bpset(*integer*) — Set a breakpoint

bpset places an unconditional breakpoint for the currently selected kernel process at the address specified by its *integer* argument. It returns the ID of the newly created breakpoint, or the nil list on error. It is simply shorthand for a call

```
bpcondset(pid, addr, {})
```

where *pid* is the global variable identifying the currently selected process, *addr* is the user-supplied address for the breakpoint, and {} is the empty list, signifying no conditions.

integer bpcondset(*pid,addr,list*) — Set conditional breakpoint

Sets a conditional breakpoint at addr for the kernel process identified by *pid*. The *list* argument is a list of operations that are executed when execution reaches *addr*. If execution results in a a non-zero value on the top of the stack, the breakpoint is taken, otherwise it is skipped. The *list* is in reverse polish notation format, and has these operations:

```
PUSH
DEREF    (pop val, push *(ulong*)val)
MASK     (pop mask, pop value, push value & mask)
EQ       (pop v1, pop v2, push v1 == v2)
NEQ      (pop v1, pop v2, push v1 != v2)
AND      (pop v1, pop v2, push v1 && v1)
OR       (pop v1, pop v2, push v1 || v2)
```

Condition lists are executed in a single pass, starting with the first command in the list, ending with the last. If a nonzero value is on the top of the stack at the end of execution, the breakpoint is taken, otherwise it is skipped.

In effect, there are two mandatory conditions, the address of the breakpoint, and the kernel process id. These two conditions must be met for the condition list to be processed. If these conditions are met, the entire condition list is processed; there is no short circuit evaluation path.

For example, given the following code fragment:

```
int i;

for(i=0; i<1000; i++) {
    ...
}
```

the following call to bpcondset() sets a conditional breakpoint to be taken when execution reaches *addr* in kernel process *pid* on the 500th iteration of the loop:

```
bpcondset(pid, addr, {bppush(i),
        bpderef(), bppush(500), bpeq()});
```

Condition List Construction

list bppush(*val*) Construct breakpoint stack

Push val onto the stack.

list bpderef() Construct breakpoint stack

Replace the value at the top of the stack with the value found at the address obtained by treating value at the top of the stack as an address. Pop the value on the top of the stack, treat it as a ulong*, and push the value at the address.

```
addr = pop();
push(*(ulong*)addr);
```

list bpmask() Construct breakpoint stack

Replace the top two values on the stack with the value obtained by masking the second value on the stack with the top of the stack.

```
mask = pop();
value = pop();
push(value & mask);
```

list bpeq() Construct breakpoint stack

Comparison of the top two values on the stack. Replace the top two values on the stack with a 1 if the values are equal, or a zero if they are not.

```
v1 = pop();
v2 = pop();
push(v1 == v2);
```

list bpneq() Construct breakpoint stack

Negative comparison of the top two values on the stack. Replace the top two values on the stack with a 0 if the values are equal, or 1 if they are not.

```
v1 = pop();
v2 = pop();
push(v1 != v2);
```

list bpand() Construct breakpoint stack

Logical and of the top two values on the stack. Replace the top two values on the stack with a 0 if both are zero, or 1 if both are nonzero.

```
v1 = pop();
v2 = pop();
push(v1 && v2);
```

list bpor() Construct breakpoint stack

Logical or of the top two values on the stack. Replace the top two values on the stack with a 1 if either is nonzero, 0 otherwise.

```
v1 = pop();
v2 = pop();
push(v1 || v2);
```

Breakpoint Status

{} bptab() List active breakpoints

Prints the list of breakpoints containing the following information in order: breakpoint number, kernel process id, breakpoint address, and the list of conditions to execute to determine if the breakpoint will be taken.

```
acid: bptab()
ID PID   ADDR             CONDITIONS
0  1   consread+0x20    0x216cc   {}
acid:
```

integer bpaddr(*id*) Address of breakpoint

Returns the address the breakpoint identified by *id* is set to trigger on.

Deleting breakpoints

{} bpdel(*id*) Delete breakpoint

Delete the breakpoint identified by *id*. Shorthand for bpconddel().

{} bpconddel(*id*) Delete conditional breakpoint

Delete the conditional breakpoint identified by the integer *id*.