Using ADB to Debug the UNIX† Kernel
Revised January, 1983

Samuel J. Leffler

William N. Joy

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

ABSTRACT

This document describes the use of extensions made to the 4.1bsd release of the VAX* UNIX debugger *adb* for the purpose of debugging the UNIX kernel. It discusses the changes made to allow standard *adb* commands to function properly with the kernel and introduces the basics necessary for users to write *adb* command scripts which may be used to augment the standard *adb* command set. The examination techniques described here may be applied to running systems, as well as the post-mortem dumps automatically created by the *savecore*(8) program after a system crash. The reader is expected to have at least a passing familiarity with the debugger command language.

---

1. INTRODUCTION

Modifications have been made to the standard VAX UNIX debugger *adb* to simplify examination of post-mortem dumps automatically generated following a system crash. These changes may also be used when examining UNIX in its normal operation. This document serves as an introduction to the **use** of these facilities, and should not be construed as a description of *how to debug the kernel*.

0.1. Invocation

When examining the UNIX kernel a new option, **−k**, should be used, e.g.

adb −k /vmunix /dev/mem

This flag causes *adb* to partially simulate the VAX virtual memory hardware when accessing the *core* file. In addition the internal state maintained by the debugger is initialized from data structures maintained by the UNIX kernel explicitly for debugging‡. A post-mortem dump may be examined in a similar fashion,

adb −k vmunix.? vmcore.?

where the appropriate version of the saved operating system image and core dump are supplied in place of ‘‘?’’.

0.2. Establishing Context

During initialization *adb* attempts to establish the context of the ‘‘currently active process’’ by examining the value of the kernel variable *masterpaddr*. This variable contains the virtual address of the process context block of the last process which was set executing by the *Swtch* routine. *Masterpaddr* normally provides sufficient information to locate the current stack frame (via the stack pointers found in the context block). By locating the VAX process context block for the process, *adb* may then perform virtual to physical address translation using that process's in-core page tables.

When examining post-mortem dumps locating the most recent stack frame of the ‘‘currently active process’’ is nontrivial. This is due to the different ways in which the VAX may save state after a nonrecoverable error. Crashes may or may not be ‘‘clean’’ (i.e. the top of the interrupt stack contains the process's kernel mode stack pointer and program counter); an ‘‘unclean’’ crash will occur, for instance, if the interrupt stack overflows. Thus, one must manually try one of two possible techniques to get a stack trace from a post-mortem dump. If the crash was clean the current stack pointer is present in the restart parameter block, at *scb−4* (or *rpb*+1fc), and the command

*(**scb**−4)$c

will generate a stack trace all the way from the kernel to the top of the user process's stack (e.g. to the *main* routine in the user process which was running). Otherwise, one must scan through the interrupt stack looking for the stack frame. This is usually indicated by a zero longword entry (the procedure call handler) followed by a longword entry with bit 29 set (indicating the call frame was generated as a result of a ‘‘calls’’ instruction).

intstack/X

Once the stack pointer has been located, the command

will generate a stack trace. An alternate method may be used when a trace of a particular

‡ If the −k flag is not used when invoking *adb* the user must explicitly calculate virtual addresses. With the **−k** option *adb* interprets page tables to automatically perform virtual to physical address translation.

process is required, see section 2.3.

2. ADB COMMAND SCRIPTS

2.1. Extending the Formatting Facilities

Once the process context has been established, the complete *adb* command set is available for interpreting data structures. In addition, a number of *adb* scripts have been created to simplify the structured printing of commonly referenced kernel data structures. The scripts normally reside in the directory */usr/lib/adb*, and are invoked with the ‘‘$<’’ operator. (A later table lists the ‘‘standard’’ scripts.)

As an example, consider the following listing which contains a dump of a faulty process’s state (our typing is shown emboldened).

```
% adb −k vmunix.17 vmcore.17
sbr 8001d064 slr d9c
p0br 800efa00 p0lr 34 p1br 7f8efe00 p1lr 1ffff2
*(intstack−4)$c
_boot() from 80004025
_boot(0,4) from 80004025
_panic(80021185) from 800057e2
_soreceive(8017478c,0) from 80007c90
_read() from 800098d7
_syscall() from 8000b6e2
_Xsyscall(3,7fffe834,258) from 80000f64
?() from c1c
?() from 26a
?(0,7fffef18,7fffef1c) from 1d3
?() from 2f
800021185/s
_icpreg+99: receive
u$<u
_u:
_u:         ksp         usp
            7fffff9c    7fffe59c
            r0          r1          r2          r3
            155c00            800237d4    80041800    3
            r4          r5          r6          r7
            0           0           11090             80041800
            r8          r9          r10         r11
            80021244    c           7fffe5b4    80000000
            ap          fp          pc          psl
            7fffffe8          7fffffa4          8000b784    d80004
            p0br        p0lr        p1br        p1lr
            800efa00    4000034     7f8efe00    1ffff2
            szpt        cmap2             sswap
            2           94000307    0
            sigc1       sigc2       sigc3
            1af03fb           fa007f02    40cbc6c
_u+78:            arg0        arg1        arg2
            3           7fffe834    258
_u+8c:            segflg error  uid   gid   ruid  rgid  procp
            0     0     4     a     4     a     80041800

_u+d4:            uap         rv1         rv2         ubase
            7ffff078          0           1           7fffe834
```

```
                count       off         cdir        rdir
                258         150         8003cf00    0
_u+f4:                pathname
                .netrc
                dirp        dino  entry pdir
                3           1395  .netrc0
7ffff11c:       ofiles
                80040818    80040818    80040818    800406b0
                800406d4    800406ec    0           0
                0           0           0           0
                0           0           0           0
                0           0           0           0

                ofileflg
                0     0     0     0     0     0     0     0
                0     0     0     0     0     0     0     0
                0     0     0     0
7ffff180:       sigs
                0           360c        1           360c
                0           0           0           aae
                0           0           0           0
                0           0           0           0
                0           0           0           0
                1           0           0           0
                0           0           0           0
                0           0           0           0

                code        ar0         prbase            prsize
                0           80000000    0           0

7ffff248:       proff       prscal      eosys sep   ttyp
                0           0           0     0     800288b4

7ffff258:       ttymin      ttymaj
                0     0
7ffff25e:       xmag        xtsiz       xdsiz       xbsiz
                3c000000    10000000    108c0000    a680000

                xssiz       entloc      relflg
                0           0           6c720000
7ffff27e:       directory
                ogin
                start       acflg fpflg cmsk tsiz  dsiz
                11688             0     12   0     160000         60000

7ffff2a2:       ssiz
                80000
```

80041800$<proc

```
80041800:       link        rlink       addr
                800237d4    0           800efde0
8004180c:       upri  pri   cpu   stat  time  nice  slp   cursig
                073   073   045   03    023   024   0     0
80041814:       sig         siga0       siga1       flag
                0           80002             45          8001
```

```
80041824:  uid    pgrp  pid    ppid   poip   szpt   tsize
           4      bb    bc     bb     0      2      1e
80041834:  dsize        ssize         rssize        maxrss
           16           6             14            3fffff
80041844:  swrss        swaddr               wchan              textp
           0            0             0             80044ee0
80041854:  clktim              p0br          xlink         ticks
           0            800efa00      80041720      22
80041864:  %cpu                       ndx    idhash        pptr
           +5.1369253545999527e−02    1c     8             80041720
```

80044ee0$<text

```
80044ee0:  daddr
           7e2          0             0             0
           0            0             0             0
           0            0             0             0

           ptdaddr      size          caddr         iptr
           352          1e            80041800      8003cfa0

           rssize swrss count ccount        flag   slptim        poip
           1a     0     02    02     042    0      0
```

The cause of the crash was a ‘‘panic’’ (see the stack trace) due to the 0 argument passed the *soreceive* routine. The majority of the dump was done to illustrate the use of two command scripts used to format kernel data structures. The ‘‘u’’ script, invoked by the command ‘‘u$<u’’, is a lengthy series of commands which pretty-prints the user vector. Likewise, ‘‘proc’’ and ‘‘text’’ are scripts used to format the obvious data structures. Let’s quickly examine the ‘‘text’’ script (the script has been broken into a number of lines for convenience here; in actuality it is a single line of text).

```
./"daddr"n12Xn\
"ptdaddr"16t"size"16t"caddr"16t"iptr"n4Xn\
"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n2x4bx++n
```

The first line produces the list of disk block addresses associated with a swapped out text segment. The ‘‘n’’ format forces a new-line character, with 12 hexadecimal integers printed immediately after. Likewise, the remaining two lines of the command format the remainder of the text structure. The expression ‘‘16t’’ causes *adb* to tab to the next column which is a multiple of 16. The last two plus operators are present to round ‘‘.’’ to the end of the text structure. This allows the user to reinvoke the format on consecutive text structures without having to be concerned about proper alignment of ‘‘.’’.

The majority of the scripts provided are of this nature. When possible, the formatting scripts print a data structure with a single format to allow subsequent reuse when interrogating arrays of structures. That is, the previous script could have been written

```
./"daddr"n12Xn
+/"ptdaddr"16t"size"16t"caddr"16t"iptr"n4Xn
+/"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n2x4bx++n
```

but then reuse of the format would have invoked only the last line of the format.

2.2. Traversing Data Structures

The *adb* command language can be used to traverse complex data structures. One such data structure, a linked list, occurs quite often in the kernel. By using *adb* variables and the normal expression operators it is a simple matter to construct a script which chains down the list printing each element along the way.

For instance, the queue of processes awaiting timer events, the callout queue, is printed with the following two scripts:

callout:

```
calltodo/"time"16t"arg"16t"func"12+
*+$<callout.next
```

callout.next:

```
./Dpp
*+>l
,#<l$<
<l$<callout.next
```

The first line of the script **callout** starts the traversal at the global symbol *calltodo* and prints a set of headings. It then skips the empty portion of the structure used as the head of the queue. The second line then invokes the script **callout.next** moving ‘‘.’’ to the top of the queue (‘‘*+’’ performs the indirection through the link entry of the structure at the head of the queue).

callout.next prints values for each column, then performs a conditional test on the link to the next entry. This test is performed as follows,

*+>l Place the value of the ‘‘link’’ in the *adb* variable ‘‘<l’’.

,#<l$< If the value stored in ‘‘<l’’ is non-zero, then the current input stream (i.e. the script **callout.next**) is terminated. Otherwise, the expression ‘‘#<l’’ will be zero, and the ‘‘$<’’ will be ignored. That is, the combination of the logical negation operator ‘‘#’’, *adb* variable ‘‘<l’’, and ‘‘$<’’ operator creates a statement of the form,

```
if (!link) exit;
```

The remaining line of **callout.next** simply reapplies the script on the next element in the linked list.

A sample *callout* dump is shown below.

```
% adb −k /vmunix /dev/mem
sbr 8001f864 slr d9c
p0br 800efa00 p0lr 8e p1br 7f8efe00 p1lr 1ffff2
$<callout
_calltodo:
_calltodo:  time     arg           func
8004ecfc:   26       0             _dzscan
8004ed0c:   8        0             _upwatch
8004ed1c:   0        0             _ip_timeo
8004ed5c:   0        0             _tcp_timeo
8004ed6c:   0        0             _rkwatch
8004ecfc:   52       0             _dzscan
8004ed2c:   68       _Syssize+70   _tmtimer
8004ed3c:   2920     0             _memenable
```

2.3. Supplying Parameters

If one is clever, a command script may use the address and count portions of an *adb* command as parameters. An example of this is the **setproc** script used to switch to the context of a process with a known process-id;

```
0t99$<setproc
```

The body of **setproc** is

```
.>4
*nproc>l
*proc>f
$<setproc.nxt
```

while **setproc.nxt** is

```
(*(<f+28))&0xffff="pid "X
,#((*(<f+28)&0xffff)-<4)$<setproc.done
<l-1>l
<f+70>f
,#<l$<
$<setproc.nxt
```

The process-id, supplied as the parameter, is stored in the variable ‘‘<4’’, the number of processes is placed in ‘‘<l’’, and the base of the array of process structures in ‘‘<f’’. **setproc.nxt** then performs a linear search through the array until it matches the process-id requested, or until it runs out of process structures to check. The script **setproc.done** simply establishes the context of the process, then exits.

2.4. Standard Scripts

The following table summarizes the command scripts currently available in the directory */usr/lib/adb*.

Standard Command Scripts		
Name	Use	Description
buf	*addr*$<**buf**	format block I/O buffer
callout	$<**callout**	print timer queue
clist	*addr*$<**clist**	format character I/O linked list
dino	*addr*$<**dino**	format directory inode
dir	*addr*$<**dir**	format directory entry
dirblk	*addr*$<**dirblk**	scan directory entries
file	*addr*$<**file**	format open file structure
fs	*addr*$<**filsys**	format in-core super block structure
findproc	*pid*$<**findproc**	find process by process id
hosts	*addr*$<**hosts**	format IMP host table entries
hosttable	*addr*$<**hosttable**	show all IMP host table entries
ifnet	*addr*$<**ifnet**	format network interface structure
ifuba	*addr*$<**ifuba**	format UNIBUS resource structure
inode	*addr*$<**inode**	format in-core inode structure
inpcb	*addr*$<**inpcb**	format internet protocol control block
iovec	*addr*$<**iovec**	format a list of *iov* structures
ipreass	*addr*$<**ipreass**	format an ip reassembly queue
mact	*addr*$<**mact**	show ‘‘active’’ list of mbuf’s
mbstat	$<**mbstat**	show mbuf statistics
mbuf	*addr*$<**mbuf**	show ‘‘next’’ list of mbuf’s
mbufs	*addr*$<**mbufs**	show a number of mbuf’s
mount	*addr*$<**mount**	format mount structure
pcb	*addr*$<**pcb**	format process context block
proc	*addr*$<**proc**	format process table entry
rawcb	*addr*$<**rawcb**	format a raw protocol control block
rtentry	*addr*$<**rtentry**	format a routing table entry
setproc	*pid*$<**setproc**	switch process context to *pid*
socket	*addr*$<**socket**	format socket structure
tcpcb	*addr*$<**tcpcb**	format TCP control block
tcpip	*addr*$<**tcpip**	format a TCP/IP packet header
tcpreass	*addr*$<**tcpreass**	show a TCP reassembly queue
text	*addr*$<**text**	format text structure
traceall	$<**traceall**	show stack trace for all processes
tty	*addr*$<**tty**	format tty structure
u	*addr*$<**u**	format user vector, including pcb
ubahd	*addr*$<**ubahd**	format a UNIBUS header structure

3. SUMMARY

The extensions made to *adb* provide basic support for debugging the UNIX kernel by eliminating the need for a user to carry out virtual to physical address translation. A collection of scripts have been written to nicely format the major kernel data structures and aid in switching between process contexts. This has been carried out with only minimal changes to the debugger.

More work is needed to provide enough information for the debugger to automatically establish context after a system crash. The system currently does not always save enough state to allow the debugger to reliably locate the stack frame just prior to an exception.

More work is also required on the user interface to *adb*. It appears the inscrutable *adb* command language has limited widespread use of much of the power of *adb*. One possibility is to provide a more comprehensible ‘‘adb frontend’’, just as *bc*(1) is used to frontend *dc*(1).

Finally, *adb* could be significantly improved if it were knowledgeable about a program’s data structures. This would eliminate the use of numeric offsets into C structures.