Using ADB to Debug the UNIX* Kernel

Samuel J. Leffler and William N. Joy

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

This document describes the facilities found in the 4.3BSD version of the VAX* UNIX debugger adb which may be used to debug the UNIX kernel. It discusses how standard adb commands may be used in examining the kernel and introduces the basics necessary for users to write adb command scripts which can augment the standard adb command set. The examination techniques described here may be applied both to running systems and 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.

Revised June 3, 1986

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.

---

*UNIX is a Trademark of Bell Laboratories.
*DEC and VAX are trademarks of Digital Equipment Corporation.

1.1. Invocation

When examining post-mortem dumps of the UNIX kernel the -k option should be used, e.g.

```
% adb -k vmunix.? vmcore.?
```

where the appropriate version of the saved operating system image and core dump are supplied in place of ``?''. 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 kernel explicitly for debugging. A running kernel may be examined in a similar fashion,

```
% adb -k /vmunix /dev/mem
```

1.2. 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 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 last currently active process can be nontrivial. This is due to the different ways in which state may be saved after a nonrecoverable error. Crashes may or may not be ``clean'' (i.e. the top of the interrupt stack contains a pointer to the process's kernel mode stack pointer and program counter); an ``unclean'' crash will occur, for instance, if the interrupt stack overflows. When adb is invoked on a post-mortem crash dump it tries to automatically establish the proper stack frame. This is done by first checking the stack pointer normally saved in the restart parameter block at rpb+1fc (or scb-4). If this value does not point to a valid stack frame, adb searches the interrupt stack looking for a valid stack frame. Should this also fail adb then searches the kernel stack located in

---

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.

the user structure associated with the last executing process. If adb is able to locate a valid stack frame using this procedure the command

```
$c
```

will generate a stack trace from the last point at which the kernel was executing on behalf of the user process all the way to the top of the user process's stack (e.g. to the main routine in the user process). Should adb be unable to locate a valid stack frame it prints a message and the current state is left undefined. When a stack trace of a particular process (other than that which was currently executing) is desired, an alternate method, described in sS2.4, should be used.

Additional information may be obtained from the kernel stack. Discussion of that subject is postponed until command scripts have been introduced; see sS2.2.

2. 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/share/adb, and are invoked with the ``$<'' operator. (A later table lists the standard scripts distributed with the system.)

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.175 vmcore.175
sbr 5868 slr 2770
p0br 5a00 p0lr 236 p1br 6600 p1lr fff0
panic: dup biodone
$c
_boot() from _boot+f3
_boot(0,0) from _panic+3a
_panic(800413d0) from _biodone+17
_biodone(800791e8) from _rxpurge+23
_rxpurge(80044754) from _rxstart+5a
_rxstart(80044754) from 80031df8
_rxintr(0) from _Xrxintr0+11
_Xrxintr0(45b01,3aaf4) from 457f
_Syssize(3aaf4) from 365a
_Syssize() from 19a8
?() from 2ff3
_Syssize(4,7fffe834) from 9cf3
```

```
_Syssize(4,7fffe834,7fffe848) from 37
?()
u$<u
_u:
_u:       ksp             usp
          7fffff94        7fffe24c
          r0              r1              r2              r3
          12e000          80044e60        800661bc        15fd1
          r4              r5              r6              r7
          13              4               80065114        16544
          r8              r9              r10             r11
          a0              80066de8        15a08           80000000
          ap              fp              pc              psl
          7fffffe8        7fffffa4        80029ed2        180000
          p0br            p0lr            p1br            p1lr
          802f5a00        4000236         7faf6600        1ffff0
          szpt            cmap2           sswap
          6               94000e59        0
_u+80:    procp           ar0             comm
          80066de8        80000000        ccom^@^@^@^
_u+9c:    arg0            arg1            arg2
          46bfc           3aefc           0
_u+bc:    uap             qsave
          7fffec9c        7fffffa4        8002a11a
_u+f8:    rv1             rv2             error   eosys
          0               3aafa           0       03
7fffed02: uid     ruid    gid     rgid
          2025    2025    10      10
7fffed0a: groups
          10      0       2       3       11      79      -1      -1
          -1      -1      -1      -1      -1      -1      -1      -1

 7fffed2c:       tsize           dsize           ssize
                 aa              18c             6
 7fffeff0:       odsize          ossize          outime
                 52              40              0
 7fffeffc:       signal
                 0               0               0               0
                 0               0               0               0
                 7a10            0               0               0
                 0               0               0               0
                 0               0               0               0
                 0               0               0               0
                 0               0               0               0
                 0               0               0               0
                 sigmask
                 0               4000            0               0
                 0               0               0               0
                 0               0               0               0
                 0               0               0               1
                 0               0               0               0
                 0               0               0               0
                 0               0               0               0
                 0               0               0               0
```

```
7ffff0fc:       onstack         sigintr         oldmask
                0               0               80002
7ffff108:       code            sigstack        onsigstack
                0               0               0
7ffff114:       ofile
                80063e40        80063e58        80064ce0        0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0
                0               0               0               0

                pofile
                0       0       0       0       0       0       0       0
                0       0       0       0       0       0       0       0
                0       0       0       0       0       0       0       0
                0       0       0       0       0       0       0       0
                0       0       0       0       0       0       0       0
                0       0       0       0       0       0       0       0
                0       0       0       0       0       0       0       0
                0       0       0       0       0       0       0       0
7ffff254:       lastfile
                2
7ffff258:  cdir            rdir            ttyp            ttyd    cmask
           80060f80        0               80056be8        106     02
7ffff268:  utime                           stime
           1               15f90           1               cf850
7ffff278:  maxrss          ixrss           idrss           isrss
           432             28250           79590           0
7ffff288:  minflt          majflt          nswap
           64              7               0
7ffff294:  inblock         oublock         msgsnd          msgrcv
           12              19              0               0
7ffff2a4:  nsignals        nvcsw           nivcsw
           0               12              22
7ffff2b0:  cru
7ffff2b0:  utime                           stime
           0               0               0               0
7ffff2c0:  maxrss          ixrss           idrss           isrss
           0               0               0               0
7ffff2d0:  minflt          majflt          nswap
           0               0               0
```

```
7ffff2dc:       inblock         oublock         msgsnd          msgrcv
                0               0               0               0
7ffff2ec:       nsignals        nvcsw           nivcsw
                0               0               0
7ffff2f8:       itimers
                0               0               0               0
                0               0               0               0
                0               0               0               0
7ffff328:       XXX
                0               0               0
7ffff334:       start                   acflag
                1985 Nov  1 21:27:18    0
7ffff340:       pr_base         pr_size         pr_off          scale
                0               0               0               0
7ffff350:       limits
                7fffffff        7fffffff        7fffffff        7fffffff
                600000          1000000         80000           1000000
                7fffffff        7fffffff        123000          123000
7ffff380:       quota           qflags
                80074a18        0
7ffff388:       nc_off          nc_inum         nc_dev  nc_time
                284             2               8       1985 Nov  1
21:27:19
7ffff398:       ni_dirp         nameiop ni_err  ni_pdir         ni_bp
                7fffe8a8        41      0       200             800606c4
7ffff3a8:       ni_base         ni_count        ni_iovec        ni_iovcnt
                0               92              7ffff3a8        1
7ffff3b8:       ni_offset       ni_segflg       ni_resid
                284             0               0
7ffff3c4:       ni_dent.d_inum  reclen  namlen  name
                19              72      9       ctm110435^@c^@^@^@
80066de8$<proc
80066de8:       link            rlink           next            prev
                80044e50        0               80067dec        8004e198
80066df8:       addr            upri    pri     cpu     stat    time
                802f65d8        0150    0150    0330    03      04
80066e01:       nice    slp     cursig          sig
                0       0       0               0
80066e08:       mask            ignore          catch
                0               0               80
80066e14:       flag            uid     pgrp    pid     ppid
                1008001         2025    11019   11045   11043
80066e20:       xstat           ru              poip    szpt    tsize
                0               0               0       6       aa
80066e30:       dsize           ssize           rssize          maxrss
                18c             6               13c             918
80066e40:       swrss           swaddr          wchan           textp
                0               6d8             0               8006b400
80066e50:       p0br            xlink           ticks
                802f5a00        0               0
80066e5c:       %cpu                            ndx     idhash  pptr
                +0.0000000000000000e+00         3ea4    106a    2e
80066e68:       cptr            osptr           ysptr
                80067dec        0               0
```

```
80066e74:       real itimer
                0               0               0               0
80066e84:       quota           0
8006b400$<text
8006b400:       forw            back
                1f30            0
                daddr
                0               0               0               0
                0               0               0               0
                0               0               2c2             aa

                ptdaddr         size            caddr           iptr
                80066de8        8005f4a0        74              10001

                rssize  swrss   count   ccount  flag    slptim  poip
                22      0       0100    031     0       0       0
```

The cause of the crash was a ``panic'' (see the stack trace) due to an inconsistency recognized inside the biodone 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 with the command ``u$<u'', is a lengthy series of commands which pretty-prints the user structure. 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).

```
        ./"forw"16t"back"n2Xn\
        "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 displays the pointers associated with the doubly linked list used in managing text segments. The second 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

```
        ./"forw"16t"back"n2Xn
        +/"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. Locating stack frames

It is frequently desirable to locate stack frames in order to examine local and register variables. In particular, frames created by a trap include saved values of all registers and the trap context, and all registers are saved upon a panic as well. Two scripts are provided for tracing stack frames. The first is capable of tracing through multiple frames, printing the information common to each. The second prints all of the information available in the stack frame after a trap. The following example illustrates their use.

```
        % adb -k vmunix.188 vmcore.188
        sbr 7068 slr 2770
        p0br 5a00 p0lr 74 p1br 5e00 p1lr fff0
        panic: Segmentation fault
        $c
        _boot() from 80029ddb
        _boot(0,0) from _panic+3a
        _panic(800447a8) from _trap+ac
        _trap() from _Xtransflt+1d
        _Xtransflt() from _Xsyscall+c
        _Xsyscall(7fffe7ac,1b6) from 514
        ?(7fffe7ac) from 4ac
        ?() from 196
        ?(2,7fffe810,7fffe81c) from 3d
        ?()
        1000$s
        *(rpb+1fc),4$<frame
7ffffe74:       handler         psr             mask
                0               0               2101
                ap              fp              pc
                7ffffec0        7ffffe9c        80029ddb        _boot+103
7ffffe9c:       handler         psr             mask
                0               0               2f00
                ap              fp              pc
                7fffff14        7ffffed0        80012de2        _panic+3a
7ffffed0:       handler         psr             mask
                0               0               2fff
                ap              fp              pc
                7fffff70        7fffff2c        8002a408        _trap+ac
```

```
7fffff2c:       handler         psr             mask
                0               0               2fff
                ap              fp              pc
                7fffffe8        7fffffa4        80001031   _Xtransflt+1d
<1$<trapframe
7fffff2c:       handler         psr             mask
                0               0               2fff
                ap              fp              pc
                7fffffe8        7fffffa4        80001031   _Xtransflt+1d
                r0              r1              r2              r3
                0               80046988        80046a00        800728db
                r4              r5              r6              r7
                800728b0        80054158        80063a60        80066ee0
                r8              r9              r10             r11
                80041b80        8               7fffe578        80000000
7fffff70:       nargs           sp              type            code
                0               7fffe560        8               2a50b6ca
                pc              (pc)            ps
                80001651        _Swtch+2b       d80008
80001651?i
_Swtch+2b:      remque  *0(r1),r2
80046988/X
_qs:
_qs:            2a50b6ca
```

The example shows a panic due to a segmentation fault. The command ``1000$s'' expands the range over which addresses will be displayed symbolically. The back trace indicates that the trap occurred four frames from the end; as the frame pointer is stored at rpb+1fc, the command ``*(rpb+1fc),4$<frame'' prints the last four stack frames; ``*(rpb+1fc)'' is the initial frame pointer, and the count determines the number of frames to print. Having located the stack frame after the trap (the frame with a return PC of Xtransflt+1d), that frame may be displayed again using the script for a trap frame. The previous frame pointer was left in register 1 by the previous script, and thus ``<1$<trapframe'' displays the state at the time of the trap. The PC at the time of the fault is shown on the last line from the script, with the faulting address listed as the code in the previous line. The instruction that caused the fault can then be examined. In this example, the instruction was a remque that used a displacement addressing mode indirecting through R1. The location to which the register points is the first of the process run queues, and its first element can be seen to be corrupted; its forward pointer, 2a50b6ca, is invalid and is the address that caused the fault.

2.3. Traversing Data Structures

The adb command language can be used to traverse complex data structures. One 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 a 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 ``#'', the adb variable ``<l'', and the ``$<'' 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
```

2.4. 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+0t52))&0xffff="pid "D
,#((*(<f+0t52)&0xffff)-<4)$<setproc.done
<l-1>l
<f+0t164>f
,#<l$<
$<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.

2.5. Standard Scripts

The following table summarizes the command scripts supplied with 2.11BSD; these scripts are found in the directory /usr/share/adb.

Standard Command

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$<dirblk	scan directory entries
dmap	addr$<dmap	format a disk-map structure
dmcstats	$<dmcstats	dump statistics for dmc0
file	addr$<file	format open file structure
filsys	addr$<filsys	format in-core super block structure
findinode	inum$<findinode	find an inode in the in-core inode table
findproc	pid$<findproc	find process by process id
frame	addr,count$<frame	trace count stack frames starting at addr
hosts	addr$<hosts	format IMP host table entries
hosttable	addr$<hosttable	show all IMP host table entries
ifaddr	addr$<ifaddr	format a net-work interface address structure
ifnet	addr$<ifnet	format network interface structure
ifuba	addr$<ifuba	format UNIBUS resource structure
imp	addr$<imp	format an IMP interface state structure in
ifaddr	addr$<in_ifaddr	format internet network addresses for an interface
inode	addr$<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
mba_device	addr$<mba_device	format an MBA device structure
mba_hd	addr$<mba_hd	format an MBA queue head
mbstat	$<mbstat	show mbuf statistics
mbuf	addr$<mbuf	show ‘next’ list of mbuf's
mbufchain	addr$<mbufchain	display a chain of mbufs queued at a socket
mbufs	addr$<mbufs	show a number of mbuf's
mount	addr$<mount	format mount structure
nameidata	addr$<nameidata	format a namei parameter block
packetchain	addr$<packetchain	format a chain of packets
pcb	addr$<pcb	format process context block
proc	addr$<proc	format process table entry
protosw	addr$<protosw	format a protocol switch entry
quota	addr$<quota	format a disk quota structure
rawcb	addr$<rawcb	format a raw protocol control block
rtentry	addr$<rtentry	format a routing table entry
rusage	addr$<rusage	format a resource usage structure
setproc	pid$<setproc	switch process context to pid
socket	addr$<socket	format socket structure
stat	addr$<stat	format a stat structure
tcpcb	addr$<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	$<traceall	show stack trace for all processes
trapframe	addr$<trapframe	format a stack frame generated by a trap
tty	addr$<tty	format tty structure

Standard	Command	
u	addr$<u	format user vector, including pcb
ubadev	addr$<ubadev	format a UBA device structure
ubahd	addr$<ubahd	format a UNIBUS header structure
unpcb	addr$<unpcb	format a UNIX domain protocol control block

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 and by automatically locating the stack frame after a system crash. A collection of scripts have been written to format the major kernel data structures and aid in switching between process contexts. These facilities have been implemented with only minimal changes to the debugger. While the symbolic debugger dbx provides facilities similar to those described here it is not yet a viable alternative to adb because dbx takes too long to read in the symbol table. As soon as this problem is corrected there will be only limited need for the facilities provided by adb.