Bug, Feature, or Code Rot? Adventures in OS Debugging

Bob Supnik, 24-Mar-2002 (updated 26-Jan-2005)

Summary

In bringing up an old operating system on a simulator, the assumption must be that any problem is the simulator's fault; after all, the operating system worked on real hardware. This assumption has not always proved to be true. Simulators on modern PC's are often significantly faster than real hardware and thus may expose race conditions or timing bugs. Simulators may exercise code paths that, in late stage operating systems, were no longer used, such as full installs. Simulators may create configurations that were not practical, due to physical or financial limitations. Finally, simulators may present late stage operating systems with hardware configurations that, while nominally supported, could in practice no longer be tested.

Timing Problems

On modern PC's, simulators for a computer architecture are often significantly faster than any real hardware that was ever built. PDP-10 simulators, for example, have been clocked at over 10 mips; the fastest DEC PDP-10 (the KL10) was 1.5 mips. Simulated devices are often much faster than their real counterparts. These speed changes can expose timing dependencies in operating system code.

A trivial case is a timing loop. Some software environments, such as console- or microcomputer-based games, are very dependent on timing loops. Timing loops also occur in system bootstraps. For example, the VAX KA655 boot code uses delay loops executing directly from boot ROM to run "slowly" as a wait loop on clock ticks. Finally, timing loops show up frequently in diagnostics.

More subtle problems occur around interrupts. Operating system code often assumes that a large amount of time elapses between initiation of an I/O operation and receipt of the completion interrupt. If the interrupt "too soon", it may be misinterpreted or lost. All versions of the RSX11M+ MSCP driver prior to V4.5 had this problem in the initialization sequence. VAX NetBSD driver has this problem during normal operations, as Kevin Handy documented in this note to the author:

> Starting at '1->', we set up a mscp packet to put the drive online. At '2->' we ping the mscp controller to take a look at it's packets. And '3->' waits up to (100*100) time units for the controller to respond with an interrupt.
>
> The problem, is that by the time it gets to '3->', the interrupt has already occurred and been processed. It's waiting for an interrupt that has already occurred, thus the timeout fails. You can see it by single stepping through the code (it suddenly jumps out of the sequence, putters around for a while, then jumps back in).

> The CPU is expecting to have enough time to set up a timeout routine before it will get a response back. It's not expecting an instant response back. You need to delay the responses from your emulated controllers for <mumble> instructions/microseconds, and then you will then get past this problem.

Ken Harrenstein found a similar problem in the disk driver in ITS.

OpenBSD has the opposite problem: if the interrupt occurs "too late", it is lost, as I documented in this note to the OpenBSD maintainers:

> 1. In rx_putonline, the code apparently does an online and waits for an interrupt:
>
> ```
> /* Poll away */
> i = bus_space_read_2(mi->mi_iot, mi->mi_iph, 0);
> if (tsleep(&rx->ra_dev.dv_unit, PRIBIO, "rxonline", 100*100))
>         rx->ra_state = DK_CLOSED;
> ```
>
> 2. But, in fact, tsleep is being short-circuited. Because of autoconfiguration, it simply opens and then closes a window for an interrupt to occur.
>
> ```
> s = splhigh();
> if (cold || panicstr) {
>         /*
>          * After a panic, or during autoconfiguration,
>          * just give interrupts a chance, then just return;
>          * don't run any other procs or panic below,
>          * in case this is the idle process and already asleep.
>          */
>         splx(safepri);
>         splx(s);
>         if (interlock != NULL && relock == 0)
>                 simple_unlock(interlock);
>         return (0);
> }
> ```
>
> I can't follow the code in detail (the compiler optimization is very good), but none of the normal path processing (storing the arguments in the process data structure) occurs. SPL is raised to 1F, then lowed to 0, then put back where it was, and 0 is returned.
>
> 3. In the meantime, the RQ is loafing along, trying to simulate the qtime polling delay, which is set at 200. tsleep returns so quickly that only about 30 instruction times have elapsed. The packet hasn't been fetched, and no interrupt has occurred. The rx_putonline call fails.
>
> 4. A bit later, the OS goes through all this again, because rxopen finds that the state is still DK_CLOSED. It calls rx_putonline, etc. But the RQ is <still> loafing around on the first poll. The OS gets into tsleep and out again, before the <first> online packet has even been fetched. So the state remains DK_CLOSED, and panic ensues.
>
> 5. You can prove this trivially: set RQ QTIME to 100 before booting, and OpenBSD 3.5 comes right up.
>
> So one answer is to set RQ QTIME down to 100, or 50, or whatever, declare victory and go home. But QTIME is set so high because <other> operating systems had timing windows. In particular, RSX11M+ V3 requires a QTIME of at least 175, or it wont boot.

> And NetBSD has timing problems as well.
>
> All of these didn't occur in 'real' life, because the CPU speeds versus the MSCP speeds were very different. The fastest PDP11 (the J11) was only about 2X the speed of the T11 in the RQDX3. But the slowest VAX was at least 3.5X, and the VAX I'm simulating is 10X faster. So the 'right' QTIME will vary, depending on the CPU/disk hardware combinations.

To address these issues, the SIMH MSCP simulator simulates ‘delays’ between initialization steps, and between initiation of an operation and completion. The delays have to be tuned experimentally to get the right values. For example, M+ requires at least 200 instructions between initialization steps, but RSTS/E can tolerate virtually no delay after the completion of step 4. PDP-11’s need “slow” operations, but VAXen need fast operations.

Finally, the changed timing of the simulation environment may expose race conditions and bugs that have lain dormant in the code. RSX11M+ has a compound bug in which a coding error in MSCP device initialization is masked, on real hardware, by the outcome of a timing race condition. If the boot device is an MSCP disk, M+ routine RVEC brings up first the controller (routine $KRBSC) and then the boot disk (routine $UCBSC) by issuing three MSCP commands:

- Set controller characteristics
- Unit online
- Get unit status

There is a bug in the MSCP driver’s handling of the get unit status command. In the interrupt handler for command completion, routine RQRCT destroys the success status code and overwrites it with 0310 (bad block replacement needed). If the MSCP disk is ‘fast’, or the driver code paths are really long, the get unit status command completes before control returns to $UCBSC. $UCBSC sees an error status and marks the disk as offline, causing the bootstrap to fail. This is what happens on the simulator with M+ V3.0.

On the other hand, if the MSCP disk is slow, or the driver code paths tighter, control returns to $UCBSC while the get unit status is still in progress. The error status code is from the successful unit online, and $UCBSC marks the boot disk as online and returns. This is what happens on the simulator with M+ V4.0 or later, and, apparently, with real hardware.

Even with the timing race falling the ‘right’ way, it requires another bug to prevent routine RVEC from seeing the erroneous status code from the get unit status. When $UCBSC returns, RVEC sees that the unit online sequence is not complete and waits for the get unit status to set a final status code. When that status (the erroneous 0310) is set, it is ignored.   RVEC only checks to see whether the disk is online. And the disk *is* online, because $UCBSC set status from the unit online command, rather than the get unit status command.

Interestingly, when the bug in RQRCT was addressed in M+ V4.0, the fix was incorrect, and the code continued to work only because of the timing race condition.

To get around this race condition, the SIMH MSCP simulator command completion delay must be tuned experimentally. M+ 3.0 requires at least 175 instructions between initiation and completion of a command.

One Good Bug Deserves Another

RSX11M+ was not the only PDP-11 operating system to benefit from bugs that cancelled each other out. RSTS/E's tape boot for the TE16/TU45/TU77 drives (driver MM:) contains a code sequence that appears, on the surface, to be clearly incorrect. RSTS/E boots from tape by rewinding the tape and reading in the tape label (!), which contains the secondary bootstraps for the various drives. On an MM: drive, it issues rewind and then read through this subroutine:

```
MMCMD:  MOV     (R5)+,(R1)      ;Issue the command
10$:    TSTB    (R1)            ;Controller ready?
        BPL     10$             ;Nope, not yet.
        TST     RHDS(R1)        ;Drive ready?
        BPL     10$             ;Nope, keep waiting.
```

Now, this should not work. TST RHDS(R1) is not testing drive ready, it's testing formatter attention. This is clear from a different copy of the routine in the boot block:

```
110$:   TSTB    (R0)            ;WAIT FOR CONTROLLER READY
        BPL     110$            ;NOT YET
        TSTB    RHDS(R0)        ;IS DRIVE READY IN RHDS?
        BPL     110$            ;NO
```

This copy tests bit <7>, which is drive ready, rather than bit <15>, which is attention. So how did the boot block code ever work? The answer is that there's *another* bug in the code, namely, the call to MMCMD to read the tape:

```
        MOV     #157000-MMBOOT,RHBA(R1) ;Load MM boot driver
        MOV     #BOOTSZ,RHWC(R1)       ;Read our extended DOS label
        CALL    MMCMD,R5        ;Issue a command to the TM02/TM03
        .WORD   71              ;The command is read
```

BOOTSZ is the size of the bootstrap in bytes, as defined by this structure:

```
.DSECT
        .BLKB   14.             ;Normal DOS label area
MUBOOT: .BLKB   1000            ;MU boot driver
MSBOOT: .BLKB   1000            ;MS boot driver
MMBOOT: .BLKB   1000            ;MM boot driver
MTBOOT: .BLKB   1000            ;MT boot driver
BOOTSZ:                         ;Length of the entire DOS label
```

But there are two problems in the calling sequence. First, RHWC is word count, not byte count. And second, it's supposed to be 2's complement. So MOV #BOOTSZ,RHWC(R1) is loading an absurdly long word count, and the record runs out before the Massbus WC does. In the real hardware, this sets frame count error (FCE), which sets attention; hence, the TST RHDS(R1) instruction exits the loop. This detail, which is not mentioned in the hardware documentation, must be simulated for RSTS/E to boot.

Unused Paths

Simulator users routinely perform full installations of operating systems onto empty disks; indeed, a full installation is one of the litmus tests for simulator success. But in real life, this path might no longer be used or tested. DEC ceased production on DECsystem-10's in the early 80's but continued to update TOPS-10 through 1988. When the last release (7.04) came out, there were no new DECsystem-10's requiring full installs, and the code path was insufficiently tested. And, in fact, it contains a bug. This problem burned Tim Stark during debug of TS10, as documented in this note to KLH10 author Ken Harrenstein:

> There's also a bug that interferes with TOPS-10 7.04 from being built correctly from scratch; that was presumably not found because no one was doing clean installs in 1988. It has to do with enumerating magtape channels or units; the code's counting loop overflows from the MTCS2 formatter select field into the Unibus address inhibit, so that the next magtape read doesn't work. SIMH got away with it because I didn't implement address inhibit, but Tim Stark got burned in TS10 because he did. (He thought the driver required it.)

Because SIMH doesn't accurately follow the hardware, it is, ironically, immune from this problem.

A more complex case is a magtape boot bug in TOPS-20 V4.1 for the KS10. The magtape bootstrap is read into low memory and then relocated to high memory for execution. For some reason, the move is done with EXCH instructions rather than conventional moves, thus replacing the low core image with the contents of high memory. The bootstrap contains the instruction WRCSTM [77B5]. After relocation of the bootstrap, the WRCSTM's address is still pointing to low core, which has been overwritten. The WRCSTM writes garbage to the CSTM, and the boot fails, as documented in this note in alt.sys.pdp10:

> The tape bootstrap moves itself into high memory with a routine that exchanges memory locations, rather than copies them. (I have no idea why.) The WRCSTM instruction in the boot references absolute address 40127, but that's been copied to high memory, and garbage (zero for the simulator) exchanged into its place. When paging is turned on, the simulator gets an age page fail error, because the CSTM is all 0's, and the age bit gets zeroed on the second page fill. Ugh. If I run the boot again, in the same core image, it works, because the contents of 40127 are already in high memory and are brought back to the right spot by the exchange.

How could such an obvious problem been overlooked? One suggestion – that the tape bootstrap of V4.1 had simply not been tested on the KS10 – was indignantly rejected by veterans of the TOPS-20 group. They insisted that the code worked on a real KS10 CPU but could not explain how.

The answer, perhaps, lies in the observation that the bootstrap succeeds the second time, because the exchange moves a copy of the bootstrap back to low memory, and the WRCSTM retrieves the correct data. On a real KS10, the front-end console had a watchdog timer. If the main CPU failed to respond with a heartbeat in a given amount of time, the console would reboot the system – without disturbing memory. The second bootstrap would succeed. From the viewpoint of anyone debugging the bootstrap process on real hardware, there would be a small tape movement, a delay, a backspace, and then a normal boot. If the tape motion wasn't noticed, the delay could be ascribed to self-test procedures in the front-end console or other "normal" delays. The system did boot; there was no need to look deeper.

Impractical Configurations

In today's computers with megabytes of memory and gigabytes of storage, the largest configuration of a historic computer represents a tiny fraction of the available resources. Simulators can create configurations that for physical or financial reasons were impractical with real hardware. For example, the SIMH PDP-15 simulator supports an RF15 fixed head disk controller with up to 8 RS09 fixed head disks. In practice, no customer would buy that many fixed head disks; instead, the customer would buy an RP15/RP02 disk pack, which provided five times the storage at lower cost.

Apparently, the maximum RF15 configuration was never tested with the PDP-15's DOS-15 operating system. The predecessor operating system, ADSS-15, had been limited to 4 RS09 disks. DOS-15 increased this to 8, but the configuration was never tested, as Hans Pufal documented in a mail message:

> The OS exits to IOPS error code 21 when it reaches a platter number 010. The problem is that with 8 platters there will never be a NED indication. I think the problem is in the OS code:

```
75072:  CLA                         ; set platter to 0
75073:  IOT 7045                    ; force controller idle, clear done
75074:  IOT 0                       ; padding
75075:  IOT 0
75076:  IOT 0

        ; Top of platter loop
75077:  IOT 7065                    ; set disk platter
75100:  IOT 0                       ; padding
75101:  IOT 0
75102:  DSSF                        ; skip on error (NED)
75103:  JMP 75106                   ; jump if disk exists
```

```
75104:  DSCD                    ; clear status
75105:  JMP 75113               ; found NED, AC equals number of platters

      ; Disk exists, inc disk and loop back if not done 8
75106:  DSCD                    ; clear status
75107:  TAD 75401   = 001       ; add 1
75110:  SAD 75722   = 010       ; compare with 010
75111:  JMP 75231               ; jmp out if disks = 8
75112:  JMP 75077               ; not 8 so go back for next disk

75113:  DAC 75072               ; store # of platters
75114:  SNA CLL                 ; skip if AC = 0
75115:  JMP 75231               ; jump to IOPS error

        ; Error path
75231:  LAC 75131
75232:  DAC* 75731
75233:  LAW 21                  ; IOPS number
75234:  JMP 75240               ; go do IOPS error
```

I think the JMP at 75111 should be a SKP.

And indeed it should. A maximum RF15 configuration, impractically expensive at the time, was never tested.

Untestable Configurations

A simulator can mimic any implementation of a computer architecture. Further, it can implement an arbitrary assemblage of peripherals. This flexibility may significantly exceed the testing capabilities available to real developers in late stage operating systems. For example, the SIMH PDP-11 simulator emulates a KDJ11A CPU with broad set of peripherals ranging from DECtape (out of production by the early 70s) to MSCP disks (still current in the early 90s). DEC in its heyday would have been hard pressed to assemble such an eclectic set of devices. Therefore, it is not surprising that by the late 90's, the skeleton crew maintaining the PDP-11 operating systems could no longer test older hardware.

This problem is evident in the behavior of RSX11M+ V4.5 autoconfigure. V4.2 correctly identifies the simulator as an LSI-11/73 (KDJ11A CPU). But V4.5 identifies it as an "M11", Mentec's 1997 re-implementation of the J-11 in gate arrays. What happened?

M+ autoconfigure implements a series of tests that act as a sieve to eliminate classes of PDP-11 processors. When the tests are done, one and only one CPU model should be flagged. The tests are very fine grained, but the KDJ11A and M11 are *almost* identical. Both respond with MFPT = 5 and maintenance ID = 20. To distinguish them, the following code was added to autoconfigure in V4.5 (as disassembled by the simulator):

```
                                  ;;; PDR7 has W bit set
131640: MOV @#172317,@#172317     ;;; write odd byte of kernel PDR7
131646: BIT #100,@#172316         ;;; is W bit still set?
131654: BEQ 131664                ;;; if eq no
131656: BIC #200,R4               ;;; if ne yes, clear J11 bit (ie, it's an M11)
131662: BR 132124
131664: BIC #20000,R4             ;;; if eq no, clear M11 bit (ie, it's a J11)
131670: BR 132056
```

This code sequence cannot work as written. On the KDJ11A, and presumably on the M11, the MOV instruction accesses an odd address and traps while fetching the source address. The trap handler simply RTI's, and the third word of the MOV is executed as an instruction ADDF F3,(PC), which is harmless. Because the PDR is not actually written, the W bit isn't cleared, and the CPU is always classified as an M11. What is going on?

The answer comes by comparison with the CPU identification code in routine SAVSIZ:

```
20$:    MOV     #KISDR7+1,R0 ;;;POINT TO KERNEL PDR7                   ;DC535
        MOVB    (R0),(R0)    ;;;WRITE THE HIGH BYTE OF THE PDR         ;DC535
        BITB    #100,-(R0)   ;;;DOES IT SHOW WRITTEN?                  ;DC535
        BNE     60$          ;;; IF NE, YES, WE HAVE AN M11            ;DC535
```

This sequence *will* work. The MOVB doesn't trap. On a KDJ11A, a write to the PDR clears the W bit, even if the PDR is mapping itself. On the M11, apparently, it does not.

How did the bug in autoconfigure go undetected? One possibility is that autoconfigure was not tested. But a more compelling hypothesis is that the developer simply didn't have a KDJ11A available for testing. The KDJ11A is a relatively rare survival as a system processor; most J11-based PDP-11 systems were built with the KDJ11B, D, or E processor modules. The developer tried the code on an M11, and it worked; he probably didn't have a KDJ11A available to see that it didn't.

A similar problem exists with the SIMH VAX simulator. It emulates a MicroVAX 3900 with a broad range of MSCP-attached storage devices: floppies, disks, CDROM's, etc. Many of the devices had a brief lifespan and then disappeared. In particular, MSCP-attached CDROM's were superceded by SCSI-attached CDROM's. By the late 90's, MSCP CDROM's had disappeared, and their behavior could not be tested.

This problem showed up as a bug in the VMS 7.3 installation procedure. VMS 7.3 allowed a read-only copy of VMS to be booted from CD. On SIMH, this procedure did not work: the bootstrap process entered mount verification and hung forever. What happened?

Read-only bootstrapping is controlled by flag DEV$M_SWL in the DEVCHR word of the unit control block (UCB). If the system mount code detects that the boot device is write-locked, it sets DEV$M_SWL, preventing any writes to the boot disk. But the boot process never reaches system mount. Instead, it enters mount verification as a side effect of establishing a connection to an MSCP controller. Unless DEV$M_SWL is already set, mount verification treats the write-locked condition as an error, outputs an error message, and loops indefinitely waiting for the write-lock to be removed.

The SCSI class driver handles this problem by setting DEV$M_SWL if it detects write-lock in an information packet. The MSCP class driver says it sets DEV$M_SWL:

```
;       RECORD_UNIT_STATUS - copy data from GET UNIT STATUS end message to UCB.
;
; Functional Description:
;
;       The supplied MSCP end message is analyzed and appropriate fields in
;       the UCB are filled in with information contained in the end message.
;
;       If the end message is shorter than MSCP$K_LEN, it is zero filled
;       to that length.  This compensates for controllers possibly passing
;       some fields back as zeros by returning short messages.  Then various
;       geometry parameters are copied or calculated from the geometry
;       information in the end message.  If the basic cylinders/tracks/sectors
;       information produced by these calulations contains any zeros, a class
;       driver bugcheck is declared.  Finally, the two write-locked bits are
;       tested.  If either is set, the DEV$M_SWL bit is set.  Otherwise, the
;       bit is not set.
```

Actually, it does something quite different:

```
    BICL    #UCB$M_MSCP_WRTP, -      ; Clear class driver write protected
            UCB$L_DEVSTS(R3)         ; flag.
    BITW    #<MSCP$M_UF_WRTPD -      ; Is the unit data loss,
             !MSCP$M_UF_WRTPH -      ; hardware, or
             !MSCP$M_UF_WRTPS>,-     ; software write protected?
            MSCP$W_UNT_FLGS(R2)
    BEQL    60$                      ; Branch if not write protected.
    BISL    #UCB$M_MSCP_WRTP, -      ; Else, set the class driver write
            UCB$L_DEVSTS(R3)         ; protected flag.
60$:
```

The driver is setting a flag in the device-dependent DEVSTS field, rather than in the device-independent DEVCHR field (the SCSI class driver sets both). Thus, mount verification regards the write-locked condition as an error and loops forever.

How did the bug go undetected? By the time read-only booting was added to VMS, all MSCP CDROM’s had long-since gone out of service. CDROM boot was tested with SCSI CDROM’s, and worked (because the SCSI driver sets

DEV$M_SWL). It could not be tested with MSCP CDROM’s, and failed.

Conclusion

In debugging a simulator, 99% of all problems that occur in bringing up an operating system will be the simulator’s fault. Occasionally, the problem will be in the operating system itself. Operating systems contain timing dependencies that simulated devices break or may not have been tested against all possible hardware configurations. They contain canceling bugs that only work due to obscure details of the hardware. Late stage operating systems suffer from inadequate staffing, incomplete test facilities, and other limitations. The result is introduction of bugs through coding mistakes or “code rot” (code breakage as a side effect of new features). Locating these problems, and tracing them to root causes, is one of the most difficult challenges in simulator debugging.

Acknowledgements

Once again, the Internet gang of historical computer enthusiasts played an indispensable role in the work documented in this paper. Doug Carman raised the initial alarms about RSX11M+’s behavior under simulation and provided access to critical sources. Robert Alan Byer showed that there were inconsistencies in several different versions of autoconfigure. Brian McCarthy, one of the stalwarts of M+ development, provided crucial insights into the autoconfigure algorithm. Tim Shoppa demonstrated the sensitivity of RSTS/E to processor and clock parameters, and John Dundas suggested how to work around the problems. Tim Stark uncovered the TOPS-10 7.04 and TOPS-20 4.01 bugs. Kevin Handy debugged the MSCP simulator issues in VAX NetBSD. Ken Harrenstein documented the ITS disk driver bug. Markus Weber and Mark Kettenis patiently coached me through OpenBSD debug. Hans Pufal recovered DOS-15 from archival DECtapes, restored it to operation, and found the RF15/RS09 bug. Andy Goldstein, stalwart of the VMS group since its inception, pointed out the controlling role of DEV$M_SWL in read-only booting and helped me trace why it was not getting set.

SIMH is on the web at http://simh.trailing-edge.com.