Odd Comments and Strange Doings in Unix

Values of Beeta

In Sixth Edition Unix, the **mv** command could produce this diagnostic:

```
values of β will give rise to dom!
```

This was noticed by some and is recorded in a WWW page or so, which are mostly copies of the same compilation of rarely encountered and striking error messages from various systems.

The actual source line in **mv.c** that produced the message was

```
write(1,"values of \016B\017 will give rise to dom!\n",37);
```

except that in the real source, the \016B\017 was written with literal ASCII SI and SO control characters. These shifted the Model 37 Teletype into its optional extended character set, and the B printed as the Greek letter beta. See below for more about rendering beta amidst changing hardware and software.

Like most of the messages recorded in these compilations, this one was produced in some situation that we considered unlikely or as result of abuse; the details don't matter. I'm recording why the phrase was selected.

The very first use of Unix in the "real business" of Bell Labs was to type and produce patent applications, and for a while in the early 1970s we had three typists busily typing away in the grotty lab on the sixth floor. One day someone came in and observed on the paper sticking out of one of the Teletypes, displayed in magnificent isolation, this ominous phrase:

```
values of β will give rise to dom!
```

It was of course obvious that the typist had interrupted a printout (generating the "!" from the **ed** editor) and moved up the paper, and that the context must have been something like "varying values of beta will give rise to domain wall movement" or some other fragment of a physically plausible patent application.

But the phrase itself was just so striking! Utterly meaningless, but it looks like what... a warning? What is "dom?"

At the same time, we were experimenting with text-to-voice software by Doug McIlroy and others, and of course the phrase was tried out with it. For whatever reason, its rendition of "give rise to dom!" accented the last word in a way that emphasized the phonetic similarity between "doom" and the first syllable of "dominance." It pronounced "beta" in the British style, "beeta." The entire occurrence became a small, shared treasure.

The phrase had to be recorded somewhere, and it was, in the v6 source. Most likely it was Bob Morris who did the deed, but it could just as easily have been Ken.

I hope that your browser reproduces the β as a Greek beta. It is written here as '& beta ;', which works on MSIE and at least some Linux browsers, but not on Netscape 4.6 (at least mine). Formerly I tried rendering it using 'fontface=symbol b /fontface' with the appropriate angle-brackets, which works on this old Netscape and MSIE, but not recent Mozillas. Sigh. If you are using an old Netscape, with an appropriate fontface, here it is: b.

/* You are not expected to understand this */

Every now and then on Usenet or elsewhere I run across a reference to a certain comment in the source code of the Sixth Edition Unix operating system.

I've even been given two sweatshirts that quote it.

Most probably just heard about it, but those who saw it in the flesh either had Sixth Edition Unix (ca. 1975) or read the annotated version of this system by John Lions (which was republished in 1996: ISBN 1-57298-013-7, Peer-to-Peer Communications).

It's often quoted as a slur on the quantity or quality of the comments in the Bell Labs research releases of Unix. Not an unfair observation in general, I fear, but in this case unjustified. The actual code and other commentary surrounding it were precisely this:

```
/*
 * Switch to stack of the new process and set up
 * his segmentation registers.
 */
retu(rp->p_addr);
sureg();
/*
 * If the new process paused because it was
 * swapped out, set the stack level to the last call
 * to savu(u_ssav).  This means that the return
 * which is executed immediately after the call to aretu
 * actually returns from the last routine which did
 * the savu.
 *
 * You are not expected to understand this.
 */
if(rp->p_flag&SSWAP) {
```

```
  rp->p_flag =& ~SSWAP;
  aretu(u.u_ssav);
 }
 /*
  * The value returned here has many subtle implications.
  * See the newproc comments.
  */
 return(1);
```

So we tried to explain what was going on. "You are not expected to understand this" was intended as a remark in the spirit of "This won't be on the exam," rather than as an impudent challenge.

The real problem is that we didn't understand what was going on either. The savu/retu mechanism for doing process exchange was fundamentally broken because it depended on switching to a previous stack frame and executing function return code in a different procedure from the one that saved the earlier state. This worked on the PDP-11 because its compiler always used the same context-save mechanism; with the Interdata compiler, the procedure return code differed depending on which registers were saved.

So, for Steve Johnson and me, trying to move the kernel for the first time to a new machine, this code was indeed on the exam. It took about a week of agonizing before we finally convinced each other that the mechanism was wrong and no fiddling with the compiler was useful. We redid the coroutine control-passing primitives altogether, and this code section, and the comment, passed into history.

Comments I do feel guilty about

Doing 32-bit multiplication and division on a 16-bit machine like the PDP-11 needs cleverness. My PDP-11 C compiler used subroutines to do long * and /. The multiplication routine was

```
/
/ 32-bit multiplication routine for fixed pt hardware.
/  Implements * operator
/ Credit to an unknown author who slipped it under the door.
.globl lmul
.globl csv, cret

lmul:
 jsr r5,csv
 mov 6(r5),r2
 sxt r1
 sub 4(r5),r1
 mov 10.(r5),r0
 sxt r3
 sub 8.(r5),r3
 mul r0,r1
 mul r2,r3
 add r1,r3
 mul r2,r0
 sub r3,r0
 jmp cret
```

which is neat, and I wasn't smart enough to figure it out. I don't feel guilty, though, because I didn't then know who suggested it and I did acknowledge the fact.

But I'll carry this one on my conscience for a while. The division routine included

```
1:
 mov r4,-(sp)
 clr r0
 div r3,r0
 mov r0,r4  /high quotient
 mov r1,r0
 mov r2,r1
 div r3,r0
 bvc 1f
 sub r3,r0  / this is the clever part
 div r3,r0
 tst r1
 sxt r1
 add r1,r0  / cannot overflow!
1:
```

I almost (or maybe even completely) figured out why it worked.

The spot on the soul is the "this is the clever part" comment.

Addendum 18 Oct 1998

Amos Shapir of nSOF (and of long memory!) just blackened (or widened) the spot a bit more in a mail message, to wit:

I gather the "almost" here is because this trick almost worked... It has a nasty bug which I had to find the hard way!

The "clever part" relies on the fact that if the "bvc 1f" is not taken, it means that the result could not fit in 16 bits; in that case the long value in r0,r1 is left unchanged. The bug is that this behavior is not documented; in later models (I found this on an 11/34) when the result does fit in 16 bits but not in 15 bits (that is, overflow for signed, but not unsigned types), the overflow bit is set, but the unsigned result does overwrite the original values -- which makes this routine provide very strange results!

A hardware story

Back around 1970-71, Unix on the PDP-11/20 ran on hardware that not only did not support virtual memory, but didn't support any kind of hardware memory mapping or protection, for example against writing over the kernel. This was a pain, because we were using the machine for multiple users. When anyone was working on a program, it was considered a courtesy to yell "A.OUT?" before trying it, to warn others to save whatever they were editing.

[A substory: at some point several were sitting around working away. Bob Morris asked, almost conversationally, "what are the arguments to ld?" Someone told him. We continued typing for the next minute, as a thought began to percolate, not quite to the top of the brain-- in other words, not quite fast enough. The terminal stopped echoing before anyone could stop and say "Hold on Bob, what is it you're trying to do?"]

We knew the PDP-11/45, which did support memory mapping and protection for the kernel and other processes, was coming, but not instantly; in anticipation, we arranged with Digital Special Systems to buy a PDP-11/20 with KS-11 add-on. This was an extra system unit bolted to the processor that made it distinguish kernel from user mode, and provided a classical PDP-10 style "hi-seg" "low-seg" memory mapping unit. I seem to recall that maybe 6 of these had been made when we ordered it.

Those who remember the early PDP-11s remember that there were no multiply, divide, or shift-by-more-than-1 instructions, and that they had an optional EAE ("extended arithmetic extension") gadget, the KE-11A, that appeared in physical memory in the I/O device space: one stored the operands here and read back the answer there (in fact at 777302 and following addresses).

One problem the KS-11 had to deal with was that ordinary programs needed to do multiplies and divides, yet shouldn't be allowed to access the I/O device space. So it included circuitry that detected just the EAE addresses, and remapped them to physical, while all other virtual addresses in user mode were mapped.

When we got the KS-11 machine, the EAE just didn't work at all. With test programming, we got bus errors, or no action from the EAE unit. But, in those days when you bought such things from Digital Special Systems, you also got the circuit drawings, so after a lot of headscratching and fiddling, they were consulted.

It turned out that on one page of the drawings, there was an address comparator for the EAE address, with an out-of-page arrow labelled "EAE ADDRESS DETECTED H". On another page, there was an in-arrow labelled "EAE ADDRESS DETECTED L". We couldn't find anything between these.

In the end, we had a visit from an embarrassed Digital Special Systems guy who found an unused inverter pin and added some white wires.

These days, such a problem would be harder to fix in the field.

Tweaked June 22, 2002