```
**********************************************************
   24136 Mon Jul 30 12:50:08 2018
new/usr/src/uts/i86pc/ml/kpti_trampolines.s
9685 KPTI %cr3 handling needs fixes
**********************************************************
______unchanged_portion_omitted_

#define SET_KERNEL_CR3(spillreg)                \
        mov     %cr3, spillreg;                 \
        mov     spillreg, %gs:CPU_KPTI_TR_CR3;  \
        mov     %gs:CPU_KPTI_KCR3, spillreg;    \
        cmp     $0, spillreg;                   \
        je      2f;                             \
        mov     spillreg, %cr3;                 \
2:

#if DEBUG
#define SET_USER_CR3(spillreg)                  \
        mov     %cr3, spillreg;                 \
        mov     spillreg, %gs:CPU_KPTI_TR_CR3;  \
        mov     %gs:CPU_KPTI_UCR3, spillreg;    \
        mov     spillreg, %cr3
#else
#define SET_USER_CR3(spillreg)                  \
        mov     %gs:CPU_KPTI_UCR3, spillreg;    \
        mov     spillreg, %cr3
#endif

#define PIVOT_KPTI_STK(spillreg)                \
        mov     %rsp, spillreg;                 \
        mov     %gs:CPU_KPTI_RET_RSP, %rsp;     \
        pushq   T_FRAMERET_SS(spillreg);        \
        pushq   T_FRAMERET_RSP(spillreg);       \
        pushq   T_FRAMERET_RFLAGS(spillreg);    \
        pushq   T_FRAMERET_CS(spillreg);        \
        pushq   T_FRAMERET_RIP(spillreg)


#define INTERRUPT_TRAMPOLINE_P(errpush) \
        pushq   %r13;                           \
        pushq   %r14;                           \
        subq    $KPTI_R14, %rsp;                \
        /* Save current %cr3. */                \
        mov     %cr3, %r14;                     \
        mov     %r14, KPTI_TR_CR3(%rsp);        \
                                                \
        cmpw    $KCS_SEL, KPTI_CS(%rsp);        \
        je      3f;                             \
1:                                              \
        /* Change to the "kernel" %cr3 */       \
        mov     KPTI_KCR3(%rsp), %r14;          \
        cmp     $0, %r14;                       \
        je      2f;                             \
        mov     %r14, %cr3;                     \
2:                                              \
        /* Get our cpu_t in %r13 */             \
        mov     %rsp, %r13;                     \
        and     $(~(MMU_PAGESIZE - 1)), %r13;   \
        subq    $CPU_KPTI_START, %r13;          \
        /* Use top of the kthread stk */        \
        mov     CPU_THREAD(%r13), %r14;         \
        mov     T_STACK(%r14), %r14;            \
        addq    $REGSIZE+MINFRAME, %r14;        \
        jmp     4f;                             \
3:                                              \
        /* Check the %rsp in the frame. */      \
```



```
        /* Is it above kernel base? */          \
        mov     kpti_kbase, %r14;               \
        cmp     %r14, KPTI_RSP(%rsp);           \
        jb      1b;                             \
        /* Use the %rsp from the trap frame */  \
        mov     KPTI_RSP(%rsp), %r14;           \
        and     $(~0xf), %r14;                  \
4:                                              \
        mov     %rsp, %r13;                     \
        /* %r14 contains our destination stk */ \
        mov     %r14, %rsp;                     \
        pushq   KPTI_SS(%r13);                  \
        pushq   KPTI_RSP(%r13);                 \
        pushq   KPTI_RFLAGS(%r13);              \
        pushq   KPTI_CS(%r13);                  \
        pushq   KPTI_RIP(%r13);                 \
        errpush;                                \
        mov     KPTI_R14(%r13), %r14;           \
        mov     KPTI_R13(%r13), %r13

#define INTERRUPT_TRAMPOLINE_NOERR              \
        INTERRUPT_TRAMPOLINE_P(/**/)

#define INTERRUPT_TRAMPOLINE                    \
        INTERRUPT_TRAMPOLINE_P(pushq KPTI_ERR(%r13))

/*
 * This is used for all interrupts that can plausibly be taken inside another
 * interrupt and are using a kpti_frame stack (so #BP, #DB, #GP, #PF, #SS).
 *
 * We also use this for #NP, even though it uses the standard IST: the
 * additional %rsp checks below will catch when we get an exception doing an
 * iret to userspace with a bad %cs/%ss.  This appears as a kernel trap, and
 * only later gets redirected via kern_gpfault().
 *
 * We check for whether we took the interrupt while in another trampoline, in
 * which case we need to use the kthread stack.
 */
#define DBG_INTERRUPT_TRAMPOLINE_P(errpush)     \
        pushq   %r13;                           \
        pushq   %r14;                           \
        subq    $KPTI_R14, %rsp;                \
        /* Check for clobbering */              \
        cmp     $0, KPTI_FLAG(%rsp);            \
        je      1f;                             \
        /* Don't worry, this totally works */   \
        int     $8;                             \
1:                                              \
        movq    $1, KPTI_FLAG(%rsp);            \
        /* Save current %cr3. */                \
        mov     %cr3, %r14;                     \
        mov     %r14, KPTI_TR_CR3(%rsp);        \
                                                \
        cmpw    $KCS_SEL, KPTI_CS(%rsp);        \
        je      4f;                             \
2:                                              \
        /* Change to the "kernel" %cr3 */       \
        mov     KPTI_KCR3(%rsp), %r14;          \
        cmp     $0, %r14;                       \
        je      3f;                             \
        mov     %r14, %cr3;                     \
3:                                              \
        /* Get our cpu_t in %r13 */             \
        mov     %rsp, %r13;                     \
        and     $(~(MMU_PAGESIZE - 1)), %r13;   \
        subq    $CPU_KPTI_START, %r13;          \
```

```
290         /* Use top of the kthread stk */        \
291         mov     CPU_THREAD(%r13), %r14;         \
292         mov     T_STACK(%r14), %r14;            \
293         addq    $REGSIZE+MINFRAME, %r14;        \
294         jmp     6f;                             \
295 4:                                              \
296         /* Check the %rsp in the frame. */      \
297         /* Is it above kernel base? */          \
298         /* If not, treat as user. */            \
299         mov     kpti_kbase, %r14;               \
300         cmp     %r14, KPTI_RSP(%rsp);           \
301         jb      2b;                             \
302         /* Is it within the kpti_frame page? */ \
303         /* If it is, treat as user interrupt */ \
304         mov     %rsp, %r13;                     \
305         and     $(~(MMU_PAGESIZE - 1)), %r13;   \
306         mov     KPTI_RSP(%rsp), %r14;           \
307         and     $(~(MMU_PAGESIZE - 1)), %r14;   \
308         cmp     %r13, %r14;                     \
309         je      2b;                             \
310         /* Were we in trampoline code? */       \
311         leaq    kpti_tramp_start, %r14;         \
312         cmp     %r14, KPTI_RIP(%rsp);           \
313         jb      5f;                             \
314         leaq    kpti_tramp_end, %r14;           \
315         cmp     %r14, KPTI_RIP(%rsp);           \
316         ja      5f;                             \
317         /* If we were, change %cr3: we might */ \
318         /* have interrupted before it did. */   \
319         mov     KPTI_KCR3(%rsp), %r14;          \
320         mov     %r14, %cr3;                     \
321 5:                                              \
322         /* Use the %rsp from the trap frame */  \
323         mov     KPTI_RSP(%rsp), %r14;           \
324         and     $(~0xf), %r14;                  \
325 6:                                              \
326         mov     %rsp, %r13;                     \
327         /* %r14 contains our destination stk */ \
328         mov     %r14, %rsp;                     \
329         pushq   KPTI_SS(%r13);                  \
330         pushq   KPTI_RSP(%r13);                 \
331         pushq   KPTI_RFLAGS(%r13);              \
332         pushq   KPTI_CS(%r13);                  \
333         pushq   KPTI_RIP(%r13);                 \
334         errpush;                                \
335         mov     KPTI_R14(%r13), %r14;           \
336         movq    $0, KPTI_FLAG(%r13);            \
337         mov     KPTI_R13(%r13), %r13

339 #define DBG_INTERRUPT_TRAMPOLINE_NOERR          \
340         DBG_INTERRUPT_TRAMPOLINE_P(/**/)

342 #define DBG_INTERRUPT_TRAMPOLINE                \
343         DBG_INTERRUPT_TRAMPOLINE_P(pushq KPTI_ERR(%r13))

345         /*
346          * These labels (_start and _end) are used by trap.c to determine if
347          * we took an interrupt like an NMI during the return process.
348          */
349 .global tr_sysc_ret_start
350 tr_sysc_ret_start:

352         /*
353          * Syscall return trampolines.
354          *
355          * These are expected to be called on the kernel %gs. tr_sysret[ql] are
```



```
356          * called after %rsp is changed back to the user value, so we have no
357          * stack to work with. tr_sysexit has a kernel stack (but has to
358          * preserve rflags, soooo).
359          */
360         ENTRY_NP(tr_sysretq)
361         cmpq    $1, kpti_enable
362         jne     1f

364         mov     %r13, %gs:CPU_KPTI_R13
365         SET_USER_CR3(%r13)
366         mov     %gs:CPU_KPTI_R13, %r13
367         /* Zero these to make sure they didn't leak from a kernel trap */
368         movq    $0, %gs:CPU_KPTI_R13
369         movq    $0, %gs:CPU_KPTI_R14
370 1:
371         swapgs
372         sysretq
373         SET_SIZE(tr_sysretq)
______unchanged_portion_omitted_


649         MK_INTR_TRAMPOLINE_NOERR(div0trap)
650         MK_DBG_INTR_TRAMPOLINE_NOERR(dbgtrap)
651         MK_DBG_INTR_TRAMPOLINE_NOERR(brktrap)
652         MK_INTR_TRAMPOLINE_NOERR(ovflotrap)
653         MK_INTR_TRAMPOLINE_NOERR(boundstrap)
654         MK_INTR_TRAMPOLINE_NOERR(invoptrap)
655         MK_INTR_TRAMPOLINE_NOERR(ndptrap)
656         MK_INTR_TRAMPOLINE(invtsstrap)
657         MK_DBG_INTR_TRAMPOLINE(segnptrap)
652         MK_INTR_TRAMPOLINE(segnptrap)
658         MK_DBG_INTR_TRAMPOLINE(stktrap)
659         MK_DBG_INTR_TRAMPOLINE(gptrap)
660         MK_DBG_INTR_TRAMPOLINE(pftrap)
661         MK_INTR_TRAMPOLINE_NOERR(resvtrap)
662         MK_INTR_TRAMPOLINE_NOERR(ndperr)
663         MK_INTR_TRAMPOLINE(achktrap)
664         MK_INTR_TRAMPOLINE_NOERR(xmtrap)
665         MK_INTR_TRAMPOLINE_NOERR(invaltrap)
666         MK_INTR_TRAMPOLINE_NOERR(fasttrap)
667         MK_INTR_TRAMPOLINE_NOERR(dtrace_ret)

669         /*
670          * These are special because they can interrupt other traps, and
671          * each other. We don't need to pivot their stacks, because they have
672          * dedicated IST stack space, but we need to change %cr3.
673          */
674         ENTRY_NP(tr_nmiint)
675         pushq   %r13
676         mov     kpti_safe_cr3, %r13
677         mov     %r13, %cr3
678         popq    %r13
679         jmp     nmiint
680         SET_SIZE(tr_nmiint)
______unchanged_portion_omitted_
```

```
**********************************************************
   50081 Mon Jul 30 12:50:09 2018
new/usr/src/uts/i86pc/ml/locore.s
9685 KPTI %cr3 handling needs fixes
**********************************************************
______unchanged_portion_omitted_

1080 #endif  /* __lint */
1081 #endif  /* !__amd64 */


1084 /*
1085  *  For stack layout, see privregs.h
1086  *  When cmntrap gets called, the error code and trap number have been pushed.
1087  *  When cmntrap_pushed gets called, the entire struct regs has been pushed.
1088  */

1090 #if defined(__lint)

1092 /* ARGSUSED */
1093 void
1094 cmntrap()
1095 {}

1097 #else   /* __lint */

1099         .globl  trap            /* C handler called below */

1101 #if defined(__amd64)

1103         ENTRY_NP2(cmntrap, _cmntrap)

1105         INTR_PUSH

1107         ALTENTRY(cmntrap_pushed)

1109         movq    %rsp, %rbp

1111         /*
1112          * - if this is a #pf i.e. T_PGFLT, %r15 is live
1113          *   and contains the faulting address i.e. a copy of %cr2
1114          *
1115          * - if this is a #db i.e. T_SGLSTP, %r15 is live
1116          *   and contains the value of %db6
1117          */

1119         TRACE_PTR(%rdi, %rbx, %ebx, %rcx, $TT_TRAP) /* Uses labels 8 and 9 */
1120         TRACE_REGS(%rdi, %rsp, %rbx, %rcx)      /* Uses label 9 */
1121         TRACE_STAMP(%rdi)               /* Clobbers %eax, %edx, uses 9 */

1123         /*
1124          * We must first check if DTrace has set its NOFAULT bit.  This
1125          * regrettably must happen before the trap stack is recorded, because
1126          * this requires a call to getpcstack() and may induce recursion if an
1127          * fbt::getpcstack: enabling is inducing the bad load.
1128          */
1129         movl    %gs:CPU_ID, %eax
1130         shlq    $CPU_CORE_SHIFT, %rax
1131         leaq    cpu_core(%rip), %r8
1132         addq    %r8, %rax
1133         movw    CPUC_DTRACE_FLAGS(%rax), %cx
1134         testw   $CPU_DTRACE_NOFAULT, %cx
1135         jnz     .dtrace_induced

1137         TRACE_STACK(%rdi)
```



```
1139         movq    %rbp, %rdi
1140         movq    %r15, %rsi
1141         movl    %gs:CPU_ID, %edx

1143         /*
1144          * We know that this isn't a DTrace non-faulting load; we can now safely
1145          * reenable interrupts.  (In the case of pagefaults, we enter through an
1146          * interrupt gate.)
1147          */
1148         ENABLE_INTR_FLAGS

1150         call    trap            /* trap(rp, addr, cpuid) handles all traps */
1151         jmp     _sys_rtt

1153 .dtrace_induced:
1154         cmpw    $KCS_SEL, REGOFF_CS(%rbp)       /* test CS for user-mode trap */
1155         jne     3f                              /* if from user, panic */

1157         cmpl    $T_PGFLT, REGOFF_TRAPNO(%rbp)
1158         je      1f

1160         cmpl    $T_GPFLT, REGOFF_TRAPNO(%rbp)
1161         je      0f

1163         cmpl    $T_ILLINST, REGOFF_TRAPNO(%rbp)
1164         je      0f

1166         cmpl    $T_ZERODIV, REGOFF_TRAPNO(%rbp)
1167         jne     4f                              /* if not PF/GP/UD/DE, panic */

1169         orw     $CPU_DTRACE_DIVZERO, %cx
1170         movw    %cx, CPUC_DTRACE_FLAGS(%rax)
1171         jmp     2f

1173         /*
1174          * If we've taken a GPF, we don't (unfortunately) have the address that
1175          * induced the fault.  So instead of setting the fault to BADADDR,
1176          * we'll set the fault to ILLOP.
1177          */
1178 0:
1179         orw     $CPU_DTRACE_ILLOP, %cx
1180         movw    %cx, CPUC_DTRACE_FLAGS(%rax)
1181         jmp     2f
1182 1:
1183         orw     $CPU_DTRACE_BADADDR, %cx
1184         movw    %cx, CPUC_DTRACE_FLAGS(%rax)    /* set fault to bad addr */
1185         movq    %r15, CPUC_DTRACE_ILLVAL(%rax)
1186                                                 /* fault addr is illegal value */
1187 2:
1188         movq    REGOFF_RIP(%rbp), %rdi
1189         movq    %rdi, %r12
1190         call    dtrace_instr_size
1191         addq    %rax, %r12
1192         movq    %r12, REGOFF_RIP(%rbp)
1193         INTR_POP
1194         jmp     tr_iret_auto
1195         /*NOTREACHED*/
1196 3:
1197         leaq    dtrace_badflags(%rip), %rdi
1198         xorl    %eax, %eax
1199         call    panic
1200 4:
1201         leaq    dtrace_badtrap(%rip), %rdi
1202         xorl    %eax, %eax
1203         call    panic
1204         SET_SIZE(cmntrap_pushed)
```

```
1205         SET_SIZE(cmntrap)
_______unchanged_portion_omitted_
```

```
**********************************************************
   18790 Mon Jul 30 12:50:10 2018
new/usr/src/uts/intel/kdi/kdi_asm.s
9685 KPTI %cr3 handling needs fixes
**********************************************************
______unchanged_portion_omitted_
```

```
 435 /*
 436  * The state of the world:
 437  *
 438  * The stack has a complete set of saved registers and segment
 439  * selectors, arranged in the kdi_regs.h order.  It also has a pointer
 440  * to our cpusave area.
 441  *
 442  * We need to save, into the cpusave area, a pointer to these saved
 443  * registers.  First we check whether we should jump straight back to
 444  * the kernel.  If not, we save a few more registers, ready the
 445  * machine for debugger entry, and enter the debugger.
 446  */

 448         ENTRY_NP(kdi_save_common_state)

 450         popq    %rdi                    /* the cpusave area */
 451         movq    %rsp, KRS_GREGS(%rdi)   /* save ptr to current saved regs */

 453         pushq   %rdi
 454         call    kdi_trap_pass
 455         testq   %rax, %rax
 456         jnz     kdi_pass_to_kernel
 455         cmpq    $1, %rax
 456         je      kdi_pass_to_kernel
 457         popq    %rax /* cpusave in %rax */

 459         SAVE_IDTGDT

 461 #if !defined(__xpv)
 462         /* Save off %cr0, and clear write protect */
 463         movq    %cr0, %rcx
 464         movq    %rcx, KRS_CR0(%rax)
 465         andq    $_BITNOT(CR0_WP), %rcx
 466         movq    %rcx, %cr0
 467 #endif

 469         /* Save the debug registers and disable any active watchpoints */

 471         movq    %rax, %r15              /* save cpusave area ptr */
 472         movl    $7, %edi
 473         call    kdi_dreg_get
 474         movq    %rax, KRS_DRCTL(%r15)

 476         andq    $_BITNOT(KDIREG_DRCTL_WPALLEN_MASK), %rax
 477         movq    %rax, %rsi
 478         movl    $7, %edi
 479         call    kdi_dreg_set

 481         movl    $6, %edi
 482         call    kdi_dreg_get
 483         movq    %rax, KRS_DRSTAT(%r15)

 485         movl    $0, %edi
 486         call    kdi_dreg_get
 487         movq    %rax, KRS_DROFF(0)(%r15)

 489         movl    $1, %edi
 490         call    kdi_dreg_get
 491         movq    %rax, KRS_DROFF(1)(%r15)
```



```
 493         movl    $2, %edi
 494         call    kdi_dreg_get
 495         movq    %rax, KRS_DROFF(2)(%r15)

 497         movl    $3, %edi
 498         call    kdi_dreg_get
 499         movq    %rax, KRS_DROFF(3)(%r15)

 501         movq    %r15, %rax      /* restore cpu save area to rax */

 503         clrq    %rbp            /* stack traces should end here */

 505         pushq   %rax
 506         movq    %rax, %rdi      /* cpusave */

 508         call    kdi_debugger_entry

 510         /* Pass cpusave to kdi_resume */
 511         popq    %rdi

 513         jmp     kdi_resume

 515         SET_SIZE(kdi_save_common_state)
______unchanged_portion_omitted_

 572         ENTRY_NP(kdi_pass_to_kernel)

 574         popq    %rdi /* cpusave */

 576         movq    $KDI_CPU_STATE_NONE, KRS_CPU_STATE(%rdi)

 573         /*
 574          * We took a trap that should be handled by the kernel, not KMDB.
 579          * Find the trap and vector off the right kernel handler.  The trap
 580          * handler will expect the stack to be in trap order, with %rip being
 581          * the last entry, so we'll need to restore all our regs.  On i86xpv
 582          * we'll need to compensate for XPV_TRAP_POP.
 575          *
 576          * We're hard-coding the three cases where KMDB has installed permanent
 577          * handlers, since after we KDI_RESTORE_REGS(), we don't have registers
 578          * to work with; we can't use a global since other CPUs can easily pass
 579          * through here at the same time.
 580          *
 581          * Note that we handle T_DBGENTR since userspace might have tried it.
 582          *
 583          * The trap handler will expect the stack to be in trap order, with %rip
 584          * being the last entry, so we'll need to restore all our regs.  On
 585          * i86xpv we'll need to compensate for XPV_TRAP_POP.
 586          *
 587          * %rax on entry is either 1 or 2, which is from kdi_trap_pass().
 588          * kdi_cmnint stashed the original %cr3 into KDIREG_CR3, then (probably)
 589          * switched us to the CPU's kf_kernel_cr3. But we're about to call, for
 590          * example:
 591          *
 592          * dbgtrap->trap()->tr_iret_kernel
 593          *
 594          * which, unlike, tr_iret_kdi, doesn't restore the original %cr3, so
 595          * we'll do so here if needed.
 596          *
 597          * This isn't just a matter of tidiness: for example, consider:
 598          *
 599          * hat_switch(oldhat=kas.a_hat, newhat=prochat)
 600          *  setcr3()
 601          *  reset_kpti()
 602          *   *brktrap* due to fbt on reset_kpti:entry
```

```
603          *
604          * Here, we have the new hat's %cr3, but we haven't yet updated
605          * kf_kernel_cr3 (so its currently kas's). So if we don't restore here,
606          * we'll stay on kas's cr3 value on returning from the trap: not good if
607          * we fault on a userspace address.
608          */
609         ENTRY_NP(kdi_pass_to_kernel)

611         popq    %rdi /* cpusave */
612         movq    $KDI_CPU_STATE_NONE, KRS_CPU_STATE(%rdi)
613         movq    KRS_GREGS(%rdi), %rsp

615         cmpq    $2, %rax
616         jne     no_restore_cr3
617         movq    REG_OFF(KDIREG_CR3)(%rsp), %r11
618         movq    %r11, %cr3

620 no_restore_cr3:
621         movq    REG_OFF(KDIREG_TRAPNO)(%rsp), %rdi

623         cmpq    $T_SGLSTP, %rdi
624         je      kdi_pass_dbgtrap
594         je      1f
625         cmpq    $T_BPTFLT, %rdi
626         je      kdi_pass_brktrap
596         je      2f
627         cmpq    $T_DBGENTR, %rdi
628         je      kdi_pass_invaltrap
598         je      3f
629         /*
630          * Hmm, unknown handler.  Somebody forgot to update this when they
631          * added a new trap interposition... try to drop back into kmdb.
632          */
633         int     $T_DBGENTR

635 #define CALL_TRAP_HANDLER(name) \
636         KDI_RESTORE_REGS(%rsp); \
637         /* Discard state, trapno, err */ \
638         addq    $REG_OFF(KDIREG_RIP), %rsp; \
639         XPV_TRAP_PUSH; \
640         jmp     %cs:name

642 kdi_pass_dbgtrap:
612 1:
643         CALL_TRAP_HANDLER(dbgtrap)
644         /*NOTREACHED*/
645 kdi_pass_brktrap:
615 2:
646         CALL_TRAP_HANDLER(brktrap)
647         /*NOTREACHED*/
648 kdi_pass_invaltrap:
618 3:
649         CALL_TRAP_HANDLER(invaltrap)
650         /*NOTREACHED*/

652         SET_SIZE(kdi_pass_to_kernel)
```

*unchanged_portion_omitted_*

```
**********************************************************
   12249 Mon Jul 30 12:50:10 2018
new/usr/src/uts/intel/kdi/kdi_idt.c
9685 KPTI %cr3 handling needs fixes
**********************************************************
______unchanged_portion_omitted_

 368 /*
 369  * We receive all breakpoints and single step traps.  Some of them, including
 370  * those from userland and those induced by DTrace providers, are intended for
 371  * the kernel, and must be processed there.  We adopt this
 372  * ours-until-proven-otherwise position due to the painful consequences of
 373  * sending the kernel an unexpected breakpoint or single step.  Unless someone
 374  * can prove to us that the kernel is prepared to handle the trap, we'll assume
 375  * there's a problem and will give the user a chance to debug it.
 376  *
 377  * If we return 2, then the calling code should restore the trap-time %cr3: that
 378  * is, it really is a kernel-originated trap.
 369  * We receive all breakpoints and single step traps.  Some of them,
 370  * including those from userland and those induced by DTrace providers,
 371  * are intended for the kernel, and must be processed there.  We adopt
 372  * this ours-until-proven-otherwise position due to the painful
 373  * consequences of sending the kernel an unexpected breakpoint or
 374  * single step.  Unless someone can prove to us that the kernel is
 375  * prepared to handle the trap, we'll assume there's a problem and will
 376  * give the user a chance to debug it.
 379  */
 380 int
 381 kdi_trap_pass(kdi_cpusave_t *cpusave)
 382 {
 383         greg_t tt = cpusave->krs_gregs[KDIREG_TRAPNO];
 384         greg_t pc = cpusave->krs_gregs[KDIREG_PC];
 385         greg_t cs = cpusave->krs_gregs[KDIREG_CS];

 387         if (USERMODE(cs))
 388                 return (1);

 390         if (tt != T_BPTFLT && tt != T_SGLSTP)
 391                 return (0);

 393         if (tt == T_BPTFLT && kdi_dtrace_get_state() ==
 394             KDI_DTSTATE_DTRACE_ACTIVE)
 395                 return (2);
 393                 return (1);

 397         /*
 398          * See the comments in the kernel's T_SGLSTP handler for why we need to
 399          * do this.
 400          */
 401 #if !defined(__xpv)
 402         if (tt == T_SGLSTP &&
 403             (pc == (greg_t)sys_sysenter || pc == (greg_t)brand_sys_sysenter ||
 404             pc == (greg_t)tr_sys_sysenter ||
 405             pc == (greg_t)tr_brand_sys_sysenter)) {
 406 #else
 407         if (tt == T_SGLSTP &&
 408             (pc == (greg_t)sys_sysenter || pc == (greg_t)brand_sys_sysenter)) {
 409 #endif
 410                 return (1);
 411         }

 413         return (0);
 414 }
______unchanged_portion_omitted_
```