## Debugging

One of the objectives of this class is to provide students with experiences writing new code for large, existing codebases. We anticipate that you may run into difficulties debugging the project code: it is often difficult to build debugging skills until you have a problem in front of you that requires them. The course staff is happy to help you with debugging, with the specific goal of helping you learn to successfully apply *scientific debugging*.

Andreas Zeller's *Debugging Book* provides an excellent guide to scientific debugging. The short version is roughly: if you can't debug an issue in the first few minutes "just by looking at it", it will be hard to keep all of the relevant information in your head at once, and a formal process to help you generate and refine guesses for *why* something is wrong can be immensely useful.

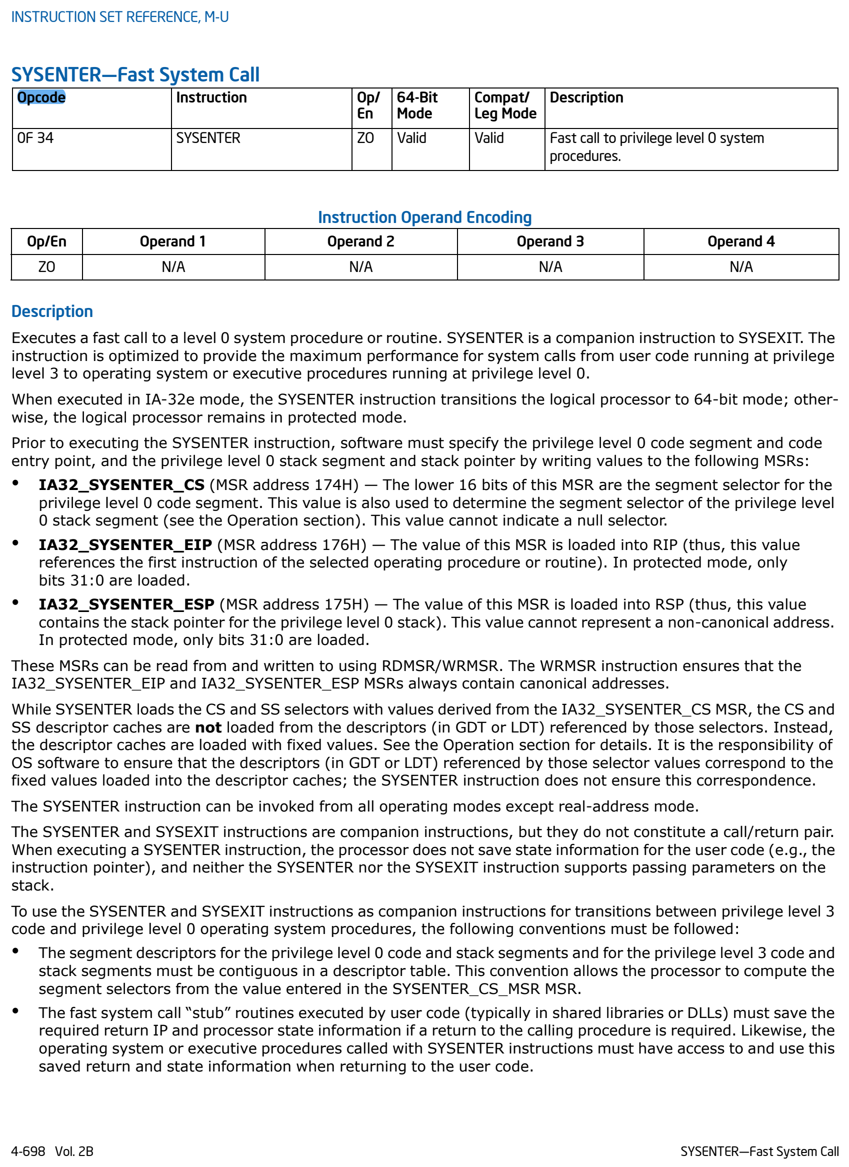
The key idea is to create a debugging note file, where you track information like:

1. What was the input/application state that caused the bug?
2. What was the behavior that I expected?
3. What was the behavior that I observed?
4. What are possible hypotheses for that behavior?
5. How have I tested those hypotheses, and what was the result?

The overall goal with hypothesis formulation is to come up with possible causes for why the bug exists. Then, as long as those hypotheses are testable, we can prove or disprove them. Most hypotheses will be along the lines of "did I make an incorrect assumption about how a library or API works." The devil is in enumerating all of the possible incorrect assumptions that you might have made, and testing them. The best way to attack these kinds of problems is to start with testing some high-level, general assumptions, and then refine them.

If you come to us for debugging help, we will ask you to answer these 5 questions, as our goal is to help you *get better at debugging* and not to simply point out bugs that we might have seen before. We are happy to discuss the problematic behavior that you are observing, possible hypotheses for why that behavior is occurring, and strategies to test those hypotheses. In the past, students have found that using a variety of strategies to test their hypotheses (e.g. using a debugger, creating a minimized test case, measured application of `console.log` statements, internet research) are useful, and we would be happy to demonstrate these. We may not be able to stay with you while you work on refining your hypotheses and fixing the bug, but would be happy to continue following up if you get stuck again.

Borrowed from Jon Bell's CS4530 22fall: https://neu-se.github.io/CS4530-Fall-2022/policies/#debugging



## SYSENTER—Fast System Call

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 34 | SYSENTER | ZO | Valid | Valid | Fast call to privilege level 0 system procedures. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| ZO | N/A | N/A | N/A | N/A |

### Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32_SYSENTER_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32_SYSENTER_EIP** (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
- **IA32_SYSENTER_ESP** (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

These MSRs can be read from and written to using RDMSR/WRMSR. The WRMSR instruction ensures that the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call "stub" routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

When shadow stacks are enabled at privilege level where SYSENTER instruction is invoked, the SSP is saved to the IA32_PL3_SSP MSR. If shadow stacks are enabled at privilege level 0, the SSP is loaded with 0. Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 17, “Control-Flow Enforcement Technology (CET)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for additional CET details.

**Instruction ordering.** Instructions following a SYSENTER may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSENTER have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

## Operation

```
IF CR0.PE = 0 OR IA32_SYSENTER_CS[15:2] = 0 THEN #GP(0); FI;

RFLAGS.VM := 0;                         (* Ensures protected mode execution *)
RFLAGS.IF := 0;                         (* Mask interrupts *)
IF in IA-32e mode
    THEN
        RSP := IA32_SYSENTER_ESP;
        RIP := IA32_SYSENTER_EIP;
ELSE
        ESP := IA32_SYSENTER_ESP[31:0];
        EIP := IA32_SYSENTER_EIP[31:0];
FI;

CS.Selector := IA32_SYSENTER_CS[15:0] AND FFFCH;
                                        (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base := 0;                           (* Flat segment *)
CS.Limit := FFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type := 11;                          (* Execute/read code, accessed *)
CS.S := 1;
CS.DPL := 0;
CS.P := 1;
IF in IA-32e mode
    THEN
        CS.L := 1;                      (* Entry is to 64-bit mode *)
        CS.D := 0;                      (* Required if CS.L = 1 *)
    ELSE
        CS.L := 0;
        CS.D := 1;                      (* 32-bit code segment*)
```

```
FI;
CS.G := 1;                              (* 4-KByte granularity *)

IF ShadowStackEnabled(CPL)
    THEN
        IF IA32_EFER.LMA = 0
            THEN IA32_PL3_SSP := SSP;
            ELSE (* adjust so bits 63:N get the value of bit N–1, where N is the CPU’s maximum linear-address width *)
                IA32_PL3_SSP := LA_adjust(SSP);
        FI;
FI;

CPL := 0;

IF ShadowStackEnabled(CPL)
    SSP := 0;
FI;
IF EndbranchEnabled(CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
    IA32_S_CET.SUPPRESS = 0
FI;

SS.Selector := CS.Selector + 8;         (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base := 0;                           (* Flat segment *)
SS.Limit := FFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type := 3;                           (* Read/write data, accessed *)
SS.S := 1;
SS.DPL := 0;
SS.P := 1;
SS.B := 1;                              (* 32-bit stack segment*)
SS.G := 1;                              (* 4-KByte granularity *)
```

## Flags Affected

VM, IF (see Operation above)

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If IA32_SYSENTER_CS[15:2] = 0. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | The SYSENTER instruction is not recognized in real-address mode. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.