Lecture 5

System Calls a.k.a.

Crossing the OS boundary safely

Obtaining System Services

- Any application requiring a resource under OS care must request for it using the proper API

- Security an issue: how to ensure the OS boundary is crossed safely?

Typical OS services

- File services: create, open, read, write, close
- Terminal I/O
- Network processing
- Memory allocation, protection etc.

`syscall()`

- Typically, system calls wrapped in library API (such as file I/O)
- To call directly, need to use **`syscall()`** library function

```
NAME
       syscall - indirect system call

SYNOPSIS
       #define _GNU_SOURCE         /* or _BSD_SOURCE or _SVID_SOURCE */
       #include <unistd.h>
       #include <sys/syscall.h>    /* For SYS_xxx definitions */

       int syscall(int number, ...);
```

syscall() example

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
    pid_t tid;

    tid = syscall(SYS_gettid);
}
```

System calls in Linux

- System calls in Linux is identified by its number

- Example: To read from an opened file requires the read system call. It is **`__NR_read`**
  - 3 in **`/usr/include/asm/unistd_32.h`** (32 bit systems)
  - 0 in **`/usr/include/asm/unistd_64.h`** (64 bit systems)

x86 system calls – the old way

- Uses a software interrupt
  - `int 0x80`
- Uses the interrupt servicing mechanism to elevate privileges to Ring 0
- But found to be slow
  - There are more things that need to be done to service interrupts in general

The new x86 way

- Uses the **`SYSENTER`**/**`SYSEXIT`** instructions

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.

In 64 bits: SYSCALL/SYSRET

- Introduced by AMD but now also supported in Intel64

Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32_LSTAR MSR always contain a canonical address.)

- Both **`SYSENTER`** and **`SYSCALL`** allows for a disciplined transition from Ring 3 to Ring 0
  - Just how to set up the context is done differently

From the Intel Manual

5.8.8 Fast System Calls in 64-Bit Mode

The SYSCALL and SYSRET instructions are designed for operating systems that use a flat memory model (segmentation is not used). The instructions, along with SYSENTER and SYSEXIT, are suited for IA-32e mode operation. SYSCALL and SYSRET, however, are not supported in compatibility mode (or in protected mode). Use CPUID to check if SYSCALL and SYSRET are available (CPUID.80000001H.EDX[bit 11] = 1).

SYSCALL is intended for use by user code running at privilege level 3 to access operating system or executive procedures running at privilege level 0. SYSRET is intended for use by privilege level 0 operating system or executive procedures for fast returns to privilege level 3 user code.

Stack pointers for SYSCALL/SYSRET are not specified through model specific registers. The clearing of bits in RFLAGS is programmable rather than fixed. SYSCALL/SYSRET save and restore the RFLAGS register.

For SYSCALL, the processor saves RFLAGS into R11 and the RIP of the next instruction into RCX; it then gets the privilege-level 0 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[47:32].
- **Target instruction pointer** — Reads a 64-bit address from IA32_LSTAR. (The WRMSR instruction ensures that the value of the IA32_LSTAR MSR is canonical.)
- **Stack segment** — Computed by adding 8 to the value in IA32_STAR[47:32].
- **Flags** — The processor sets RFLAGS to the logical-AND of its current value with the complement of the value in the IA32_FMASK MSR.

When SYSRET transfers control to 64-bit mode user code using REX.W, the processor gets the privilege level 3 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[63:48] + 16.
- **Target instruction pointer** — Copies the value in RCX into RIP.
- **Stack segment** — IA32_STAR[63:48] + 8.
- **EFLAGS** — Loaded from R11.

When SYSRET transfers control to 32-bit mode user code using a 32-bit operand size, the processor gets the privilege level 3 target code segment, instruction pointer, stack segment, and flags as follows:

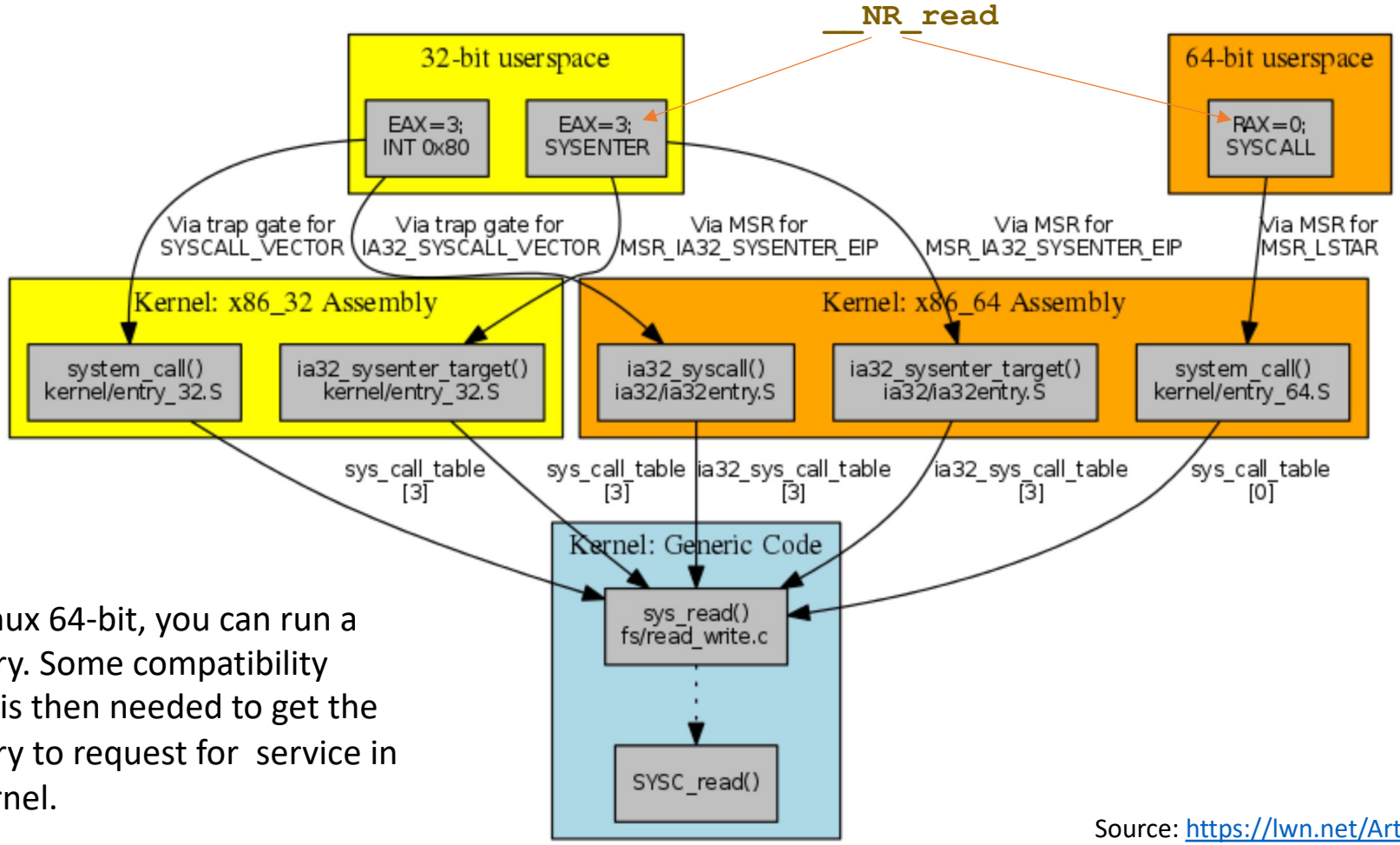
- **Target code segment** — Reads a non-NULL selector from IA32_STAR[63:48].
- **Target instruction pointer** — Copies the value in ECX into EIP.
- **Stack segment** — IA32_STAR[63:48] + 8.
- **EFLAGS** — Loaded from R11.

It is the responsibility of the OS to ensure the descriptors in the GDT/LDT correspond to the selectors loaded by SYSCALL/SYSRET (consistent with the base, limit, and attribute values forced by the instructions).

See Figure 5-14 for the layout of IA32_STAR, IA32_LSTAR and IA32_FMASK.

5-22 Vol. 3A

In Linux



Note: In Linux 64-bit, you can run a 32-bit binary. Some compatibility processing is then needed to get the 32-bit binary to request for service in a 64-bit kernel.

Source: https://lwn.net/Articles/604515/

In Linux

```
/* Usage: long syscall (syscall_number, arg1, arg2, arg3, arg4, arg5, arg6)
   We need to do some arg shifting, the syscall_number will be in
   rax.  */


        .text
ENTRY (syscall)
        movq %rdi, %rax         /* Syscall number -> rax.  */
        movq %rsi, %rdi         /* shift arg1 - arg5.  */
        movq %rdx, %rsi
        movq %rcx, %rdx
        movq %r8, %r10
        movq %r9, %r8
        movq 8(%rsp),%r9        /* arg6 is on the stack.  */
        syscall                 /* Do the system call.  */
        cmpq $-4095, %rax       /* Check %rax for error.  */
        jae SYSCALL_ERROR_LABEL /* Jump to error handler if error.  */
        ret                     /* Return to caller.  */

PSEUDO_END (syscall)
```

glibc-2.29/sysdeps/unix/sysv/linux/x86_64/syscall.S

In Linux

arch/x86/kernel/cpu/common.c

```
void syscall_init(void)
{
        wrmsr(MSR_STAR, 0, (__USER32_CS << 16) | __KERNEL_CS);
        wrmsrl(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);
```

```
#include <linux/bits.h>

/*
 * CPU model specific register (MSR) numbers.
 *
 * Do not add new entries to this file unless the definitions are shared
 * between multiple compilation units.
 */

/* x86-64 specific MSRs */
#define MSR_EFER                0xc0000080 /* extended feature register */
#define MSR_STAR                0xc0000081 /* legacy mode SYSCALL target */
#define MSR_LSTAR               0xc0000082 /* long mode SYSCALL target */
#define MSR_CSTAR               0xc0000083 /* compat mode SYSCALL target */
#define MSR_SYSCALL_MASK        0xc0000084 /* EFLAGS mask for syscall */
#define MSR_FS_BASE             0xc0000100 /* 64bit FS base */
#define MSR_GS_BASE             0xc0000101 /* 64bit GS base */
#define MSR_KERNEL_GS_BASE      0xc0000102 /* SwapGS GS shadow */
#define MSR_TSC_AUX             0xc0000103 /* Auxiliary TSC */
```

arch/x86/include/asm/msr-index.h

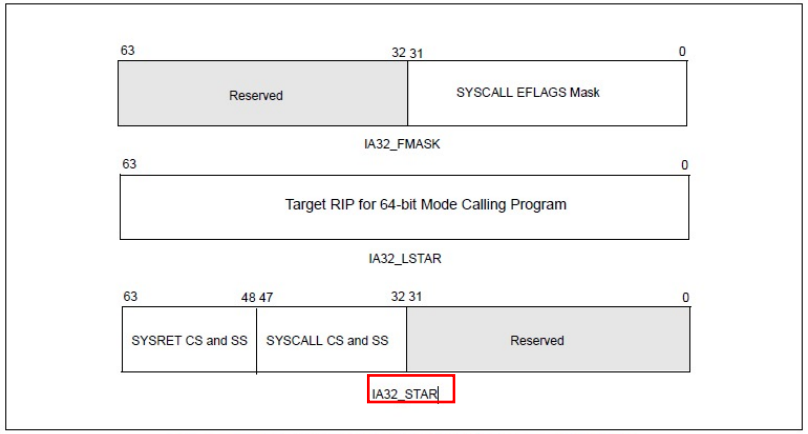


Figure 5-14. MSRs Used by SYSCALL and SYSRET

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
C000_0081H		IA32_STAR	System Call Target Address (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0082H		IA32_LSTAR	IA-32e Mode System Call Target Address (R/W) Target RIP for the called procedure when SYSCALL is executed in 64-bit mode.	If CPUID.80000001:EDX.[29] = 1

A small digression: x86 Model Specific Registers

- Special control “registers” in the x86 instruction set used for debugging, execution tracing, computer performance monitoring, and operating certain CPU features

- They may or may not be implemented in the processor you have at hand
  - Need to check the model

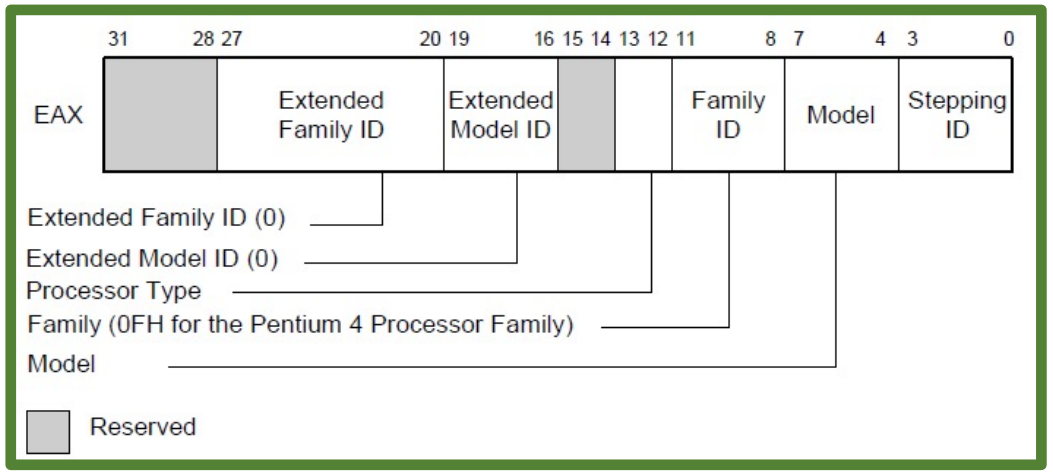
Using the MSR

- Use **`rdmsr`** to read a MSR
  - EDX:EAX = MSR[ECX]
- Use **`wrmsr`** to write a MSR
  - MSR[ECX] = EDX:EAX
- Must first confirm a particular MSR's existence using the (complicated) **`cpuid`** instruction

Digression of a digression: CPUID

- Bit 21 (ID bit) of EFLAGS must be 1 to indicate support for CPUID
- Put a value in EAX for a query code
- Values returned in various registers (read the manual)

Input EAX = 1



Linux system calls

- A table of system call service routines is maintained by the kernel
  - `sys_call_table[]` in arch/x86/entry
    - Note: it is autogenerated at kernel build time from `arch/x86/entry/syscalls/syscall_{32|64}.tbl`
- The index to the table is the assigned system call number
- `arch/x86/entry/common.c` contains the dispatch code – after SYSENTER/SYSCALL
  - Wrapped in a bit of assembly code in `entry_{32|64}.S`

Summary

- There are only two ways to enter the kernel
  - Via interrupt
  - Using SYSCALL/SYSENTER
  - Important difference: a system call always transits from user to kernel mode, while an interrupt/exception/trap can happen even inside Ring 0
- In 64-bit, Linux expects user to enter the kernel only by the use SYSCALL instructions
  - SYSENTER is used only in 32-bit Linux

Important digressions

Stacks everywhere

- The modern Linux kernel is multithreaded
  - Need stacks to operate
  - No stack, no procedure call
  - If use user stack, security risk

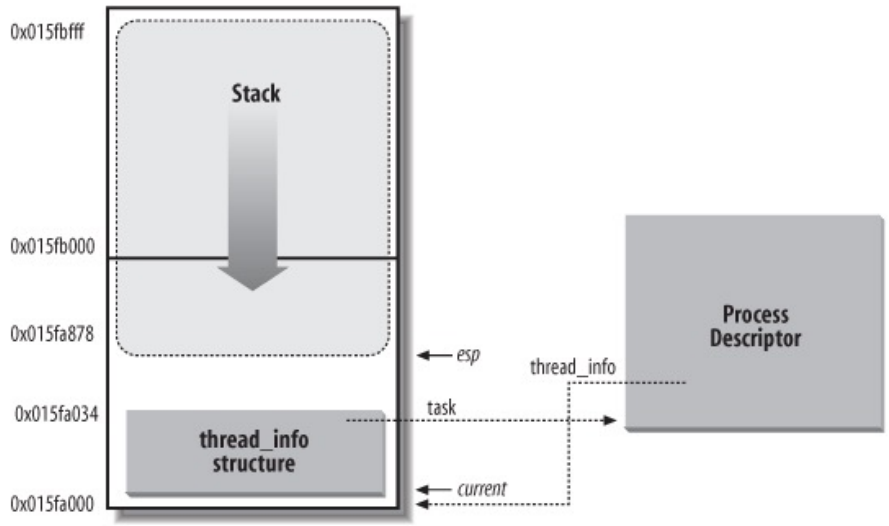
- Interrupts and stacks can happen any time

The stacks used by the kernel

- Per thread kernel stack
- Entry trampoline stack
- Interrupt Stack
- Hard IRQ Stack
- Soft IRQ Stack

Per thread kernel stack

- 16KB
- It is in the `task_struct` of each process
- Grows towards the `thread_info` structure



Task State Segment

- Intel architecture provisions for OS tasks
  - Hardware task switching not supported in 64 bit mode
- Consists of pointers to stacks
  - 7 IST stacks for interrupt service routine usage
  - One for each protection ring
    - RSP0 – entry trampoline stack
    - RSP1 – current top of kernel stack
    - RSP2 – scratch to contain user stack pointer on SYSCAL_

Although hardware task-switching is not supported in 64-bit mode, a 64-bit task state segment (TSS) must exist. Figure 7-11 shows the format of a 64-bit TSS. The TSS holds information important to 64-bit mode and that is not directly related to the task-switch mechanism. This information includes:

- **RSPn** — The full 64-bit canonical forms of the stack pointers (RSP) for privilege levels 0-2.
- **ISTn** — The full 64-bit canonical forms of the interrupt stack table (IST) pointers.
- **I/O map base address** — The 16-bit offset to the I/O permission bit map from the 64-bit TSS base.

The operating system must create at least one 64-bit TSS after activating IA-32e mode. It must execute the LTR instruction (in 64-bit mode) to load the TR register with a pointer to the 64-bit TSS responsible for both 64-bit-mode programs and compatibility-mode programs.

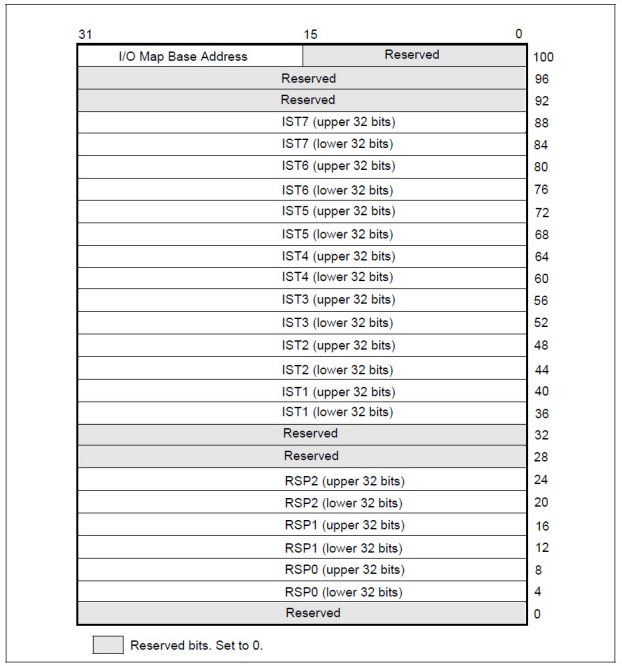


Figure 7-11. 64-Bit TSS Format

TSS in Linux

- **`arch/x86/include/asm/cpu_entry_area.h`**
  - Defines the (per) CPU entry area that includes the TSS
- **`arch/x86/include/asm/processor.h`**
  - Actual definition of the TSS
- Linux only has one TSS for each CPU and uses them for all tasks

Page Table Isolation

- Full set of kernel page tables use to coexist with user page tables
- Potential security loophole!
- Two separate sets (duplicates) of kernel page tables
  - The full one
  - A minimal one in the user space just sufficient to transit to the kernel
- Implemented on Linux 4.15 onwards

Details of SYSCALL

Hardware execution

What the Intel manual says:

Operation

```
IF (CS.L ≠ 1 ) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD;
FI;

RCX := RIP;                          (* Will contain address of next instruction *)
RIP := IA32_LSTAR;
R11 := RFLAGS;
RFLAGS := RFLAGS AND NOT(IA32_FMASK);

CS.Selector := IA32_STAR[47:32] AND FFFCH (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base := 0;                        (* Flat segment *)
```

4-680 Vol. 2B SYSCALL—Fast System Call

arch/x86/kernel/cpu/common.c:1752

```
void syscall_init(void)
{
        wrmsr(MSR_STAR, 0, (__USER32_CS << 16) | __KERNEL_CS);
        wrmsrl(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);
```

Model Specific Registers (MSR)

- MSRs are used to control and report on processor performance
- Virtually all MSRs handle system related functions and are not accessible to an application program
- One exception to this rule is the time-stamp counter
- Accessed using **RDMSR** and **WRMSR** privileged instructions

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name / Bit Fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
C000_0081H		IA32_STAR	System Call Target Address (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0082H		IA32_LSTAR	IA-32e Mode System Call Target Address (R/W) Target RIP for the called procedure when SYSCALL is executed in 64-bit mode.	If CPUID.80000001:EDX.[29] = 1
C000_0083H		IA32_CSTAR	IA-32e Mode System Call Target Address (R/W) Not used, as the SYSCALL instruction is not recognized in compatibility mode.	If CPUID.80000001:EDX.[29] = 1
C000_0084H		IA32_FMASK	System Call Flag Mask (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0100H		IA32_FS_BASE	Map of BASE Address of FS (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0101H		IA32_GS_BASE	Map of BASE Address of GS (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0102H		IA32_KERNEL_GS_BASE	Swap Target of BASE Address of GS (R/W)	If CPUID.80000001:EDX.[29] = 1
C000_0103H		IA32_TSC_AUX	Auxiliary TSC (RW)	If CPUID.80000001H: EDX[27] = 1 or CPUID.(EAX=7,ECX=0):ECX[bit 22] = 1
		31:0	AUX: Auxiliary signature of TSC.	
		63:32	Reserved	

arch/x86/entry/entry_64.S

```
/*
 * 64-bit SYSCALL instruction entry. Up to 6 arguments in registers.
 *
 * This is the only entry point used for 64-bit system calls.  The
 * hardware interface is reasonably well designed and the register to
 * argument mapping Linux uses fits well with the registers that are
 * available when SYSCALL is used.
 *
 * SYSCALL instructions can be found inlined in libc implementations as
 * well as some other programs and libraries.  There are also a handful
 * of SYSCALL instructions in the vDSO used, for example, as a
 * clock_gettimeofday fallback.
 *
 * 64-bit SYSCALL saves rip to rcx, clears rflags.RF, then saves rflags to r11,
 * then loads new ss, cs, and rip from previously programmed MSRs.
 * rflags gets masked by a value from another MSR (so CLD and CLAC
 * are not needed). SYSCALL does not save anything on the stack
 * and does not change rsp.
 *
 * Registers on entry:
 * rax  system call number
 * rcx  return address
 * r11  saved rflags (note: r11 is callee-clobbered register in C ABI)
 * rdi  arg0
 * rsi  arg1
 * rdx  arg2
 * r10  arg3 (needs to be moved to rcx to conform to C ABI)
 * r8   arg4
 * r9   arg5
 * (note: r12-r15, rbp, rbx are callee-preserved in C ABI)
 *
 * Only called from user space.
 *
 * When user can change pt_regs->foo always force IRET. That is because
 * it deals with uncanonical addresses better. SYSRET has trouble
 * with them due to bugs in both AMD and Intel CPUs.
 */
```

Pt_regs is the structure for saving all registers.
Defined in **arch/x86/include/uapi/asm/ptrace.h**

arch/x86/entry/entry_64.S

```
SYM_CODE_START(entry_SYSCALL_64)
        UNWIND_HINT_EMPTY

        swapgs
        /* tss.sp2 is scratch space. */
        movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
        SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
        movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)

        /* Construct struct pt_regs on stack */
        pushq   $__USER_DS                              /* pt_regs->ss */
        pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)       /* pt_regs->sp */
        pushq   %r11                                    /* pt_regs->flags */
        pushq   $__USER_CS                              /* pt_regs->cs */
        pushq   %rcx                                    /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
        pushq   %rax                                    /* pt_regs->orig_ax */

        PUSH_AND_CLEAR_REGS rax=$-ENOSYS

        /* IRQs are off. */
        movq    %rax, %rdi
        movq    %rsp, %rsi
        call    do_syscall_64           /* returns with IRQs disabled */
```

Code written in assembly can do funky things to the stack. This makes it hard for debugging tools to follow the stack frame properly as they don't follow the proper calling convention. This is a macro that gives "hints" to such debuggers.

See: DWARF debugging file format standard specifications: https://dwarfstd.org/

arch/x86/entry/entry_64.S

```
SYM_CODE_START(entry_SYSCALL_64)
        UNWIND_HINT_EMPTY

        swapgs
        /* tss.sp2 is scratch space. */
        movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
        SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
        movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)

        /* Construct struct pt_regs on stack */
        pushq   $__USER_DS                              /* pt_regs->ss */
        pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)       /* pt_regs->sp */
        pushq   %r11                                    /* pt_regs->flags */
        pushq   $__USER_CS                              /* pt_regs->cs */
        pushq   %rcx                                    /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
        pushq   %rax                                    /* pt_regs->orig_ax */

        PUSH_AND_CLEAR_REGS rax=$-ENOSYS

        /* IRQs are off. */
        movq    %rax, %rdi
        movq    %rsp, %rsi
        call    do_syscall_64           /* returns with IRQs disabled */
```

GS segment register is used for quick access to the per CPU region so as to get the per-CPU variables quickly .

FS and GS in Intel 64 bit mode

- In 64-bit mode, FS and GS segment registers' base address is 64 bits
  - Full segmentation is not available but you can use FS and GS as 64 bit base registers
  - Use in Linux:
    - FS points to thread local storage
    - GS points to per CPU data structure

- New 64-bit instructions to write to FS and GS segment base
  - **`wrfsbase`**, **`wrgsbase`**

`swapgs`

- GS is associated with a special control register MSR at address 0xC0000102
  - Intel manual calls this the IA32_KERNEL_GS_BASE MSR
  - Only accessible in Ring 0

- **`swapgs`** – a privileged instruction that will swap the current value of GS with IA32_KERNEL_GS_BASE MSR

arch/x86/entry/entry_64.S

```
SYM_CODE_START(entry_SYSCALL_64)
        UNWIND_HINT_EMPTY

        swapgs
        /* tss.sp2 is scratch space. */
        movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
        SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
        movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)

        /* Construct struct pt_regs on stack */
        pushq   $__USER_DS                              /* pt_regs->ss */
        pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)       /* pt_regs->sp */
        pushq   %r11                                    /* pt_regs->flags */
        pushq   $__USER_CS                              /* pt_regs->cs */
        pushq   %rcx                                    /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
        pushq   %rax                                    /* pt_regs->orig_ax */

        PUSH_AND_CLEAR_REGS rax=$-ENOSYS

        /* IRQs are off. */
        movq    %rax, %rdi
        movq    %rsp, %rsi
        call    do_syscall_64           /* returns with IRQs disabled */
```

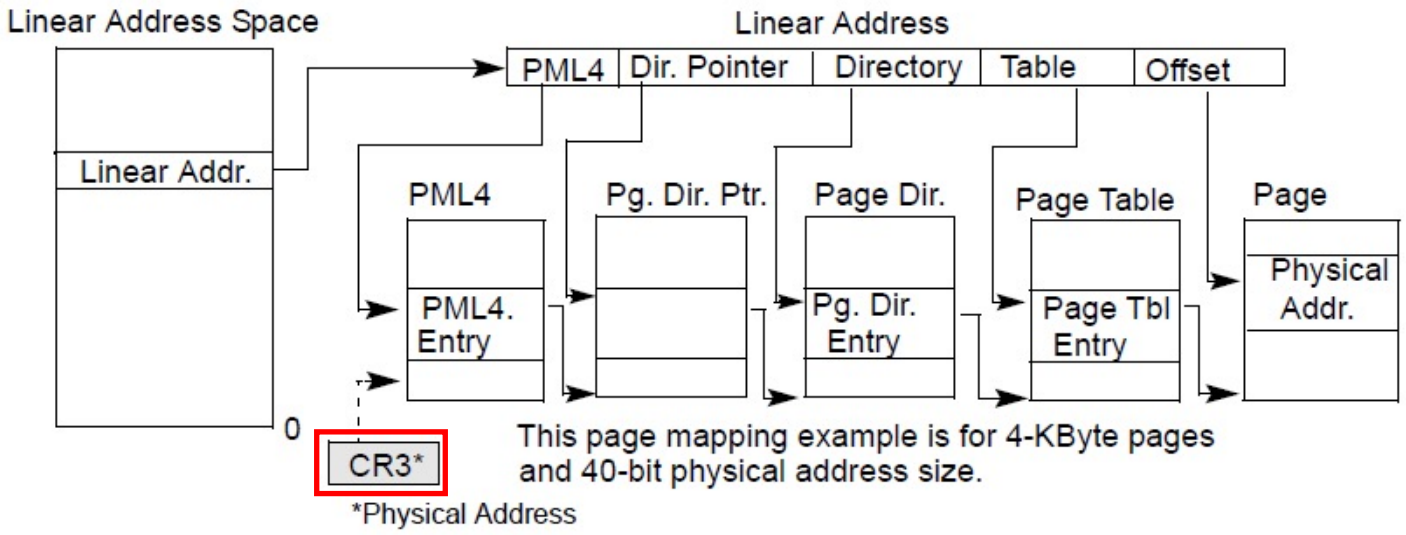
saves the user mode stack pointer into a scratch location, namely the **sp2** of the TSS which is also not used since Ring 2 is not used.

arch/x86/entry/entry_64.S

```
SYM_CODE_START(entry_SYSCALL_64)
        UNWIND_HINT_EMPTY

        swapgs
        /* tss.sp2 is scratch space. */
        movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
        SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
        movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)

        /* Construct struct pt_regs on stack */
        pushq   $__USER_DS                              /* pt_regs->ss */
        pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)       /* pt_regs->sp */
        pushq   %r11                                    /* pt_regs->flags */
        pushq   $__USER_CS                              /* pt_regs->cs */
        pushq   %rcx                                    /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
        pushq   %rax                                    /* pt_regs->orig_ax */

        PUSH_AND_CLEAR_REGS rax=$-ENOSYS

        /* IRQs are off. */
        movq    %rax, %rdi
        movq    %rsp, %rsi
        call    do_syscall_64           /* returns with IRQs disabled */
```

Restore the full kernel page table.
Use `%rsp` as a scratch since it was already saved up.

Recall: Paging and CR3



arch/x86/entry/entry_64.S

```
SYM_CODE_START(entry_SYSCALL_64)
        UNWIND_HINT_EMPTY

        swapgs
        /* tss.sp2 is scratch space. */
        movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
        SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
        movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)

        /* Construct struct pt_regs on stack */
        pushq   $__USER_DS                              /* pt_regs->ss */
        pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)       /* pt_regs->sp */
        pushq   %r11                                    /* pt_regs->flags */
        pushq   $__USER_CS                              /* pt_regs->cs */
        pushq   %rcx                                    /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
        pushq   %rax                                    /* pt_regs->orig_ax */

        PUSH_AND_CLEAR_REGS rax=$-ENOSYS

        /* IRQs are off. */
        movq    %rax, %rdi
        movq    %rsp, %rsi
        call    do_syscall_64           /* returns with IRQs disabled */
```

Switch to true kernel stack.

arch/x86/entry/entry_64.S

```
SYM_CODE_START(entry_SYSCALL_64)
        UNWIND_HINT_EMPTY

        swapgs
        /* tss.sp2 is scratch space. */
        movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
        SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
        movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)

        /* Construct struct pt_regs on stack */
        pushq   $__USER_DS                              /* pt_regs->ss */
        pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)       /* pt_regs->sp */
        pushq   %r11                                    /* pt_regs->flags */
        pushq   $__USER_CS                              /* pt_regs->cs */
        pushq   %rcx                                    /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
        pushq   %rax                                    /* pt_regs->orig_ax */

        PUSH_AND_CLEAR_REGS rax=$-ENOSYS

        /* IRQs are off. */
        movq    %rax, %rdi
        movq    %rsp, %rsi
        call    do_syscall_64           /* returns with IRQs disabled */
```

Saves (some) registers to pt_regs.

arch/x86/entry/entry_64.S

```
SYM_CODE_START(entry_SYSCALL_64)
        UNWIND_HINT_EMPTY

        swapgs
        /* tss.sp2 is scratch space. */
        movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
        SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
        movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)

        /* Construct struct pt_regs on stack */
        pushq   $__USER_DS                              /* pt_regs->ss */
        pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)       /* pt_regs->sp */
        pushq   %r11                                    /* pt_regs->flags */
        pushq   $__USER_CS                              /* pt_regs->cs */
        pushq   %rcx                                    /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
        pushq   %rax                                    /* pt_regs->orig_ax */

        PUSH_AND_CLEAR_REGS rax=$-ENOSYS

        /* IRQs are off. */
        movq    %rax, %rdi
        movq    %rsp, %rsi
        call    do_syscall_64           /* returns with IRQs disabled */
```

Call the service dispatcher.

arch/x86/entry/common.c

```
__visible noinstr void do_syscall_64(unsigned long nr, struct pt_regs *regs)
{
        nr = syscall_enter_from_user_mode(regs, nr);

        instrumentation_begin();
        if (likely(nr < NR_syscalls)) {
                nr = array_index_nospec(nr, NR_syscalls);
                regs->ax = sys_call_table[nr](regs);
```

Speeding up system calls (1)

vsyscall

vsyscall

- Certain system calls can get called very often
- How to speed it up even faster?
  - Key: do not actually enter the kernel

Executing system call in userspace

- Linux kernel maps a page containing some kernel variables and implementation of some system calls into the user space
  - Key: **read only**!

```
[wongwf@deva vsyscall]$ grep vsyscall /proc/self/maps
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Nuts and bolts

- Mapping of the **`vsyscall`** page occurs in the **`map_vsyscall`** function that is defined in the **`arch/x86/entry/vsyscall/vsyscall_64.c`**

- This is called during kernel initialization

Read only

- Used to be done for `gettimeofday()`, `time()` and `getcpu()`
  - All read only functions

- The vsyscall page contains the variables involved and the small amount of (kernel) code to read them
  - Page is readable, executable but not writable

Deprecated!

- Now deemed too dangerous!
  - Exposing a kernel physical page to the all user processes at a fixed known address

Speeding up system calls (2)

Virtual Dynamic Shared Object

vDSO

- Also same idea as vsyscall but allow linker to do address space randomization (ASR) and place the page anywhere in the virtual space

- Example from two different processes:

```
[wongwf@deva ~]$ grep vdso /proc/1233/maps
7ffc8aff5000-7ffc8aff6000 r-xp 00000000 00:00 0                          [vdso]
[wongwf@deva ~]$ grep vdso /proc/29770/maps
7fffd09eb000-7fffd09ec000 r-xp 00000000 00:00 0                          [vdso]
```

vDSO object

No associated file object

```
[wongwf@asura vsyscall]$ ldd /bin/uname
        linux-vdso.so.1 =>  (0x00007ffc62dcc000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fa49f2d7000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fa49f6a5000)
[wongwf@asura vsyscall]$ ldd /bin/uname
        linux-vdso.so.1 =>  (0x00007fff896ea000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f8362497000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f8362865000)
[wongwf@asura vsyscall]$ ldd /bin/uname
        linux-vdso.so.1 =>  (0x00007ffc9cbb0000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f6040d31000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f60410ff000)
[wongwf@asura vsyscall]$ ldd /bin/uname
        linux-vdso.so.1 =>  (0x00007ffc50f35000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fa0ae597000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fa0ae965000)
[wongwf@asura vsyscall]$ ldd /bin/uname
        linux-vdso.so.1 =>  (0x00007ffdff1ac000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f2982126000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f29824f4000)
```

vDSO object

```
[wongwf@asura vsyscall]$ ldd /bin/uname
        linux-vdso.so.1 =>  (0x00007ffc62dcc000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fa49f2d7000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fa49f6a5000)
[wongwf@asura vsyscall]$ ldd /bin/uname
        linux-vdso.so.1 =>  (0x00007fff896ea000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f8362497000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f8362865000)
[wongwf@asura vsyscall]$ ldd /bin/uname
        linux-vdso.so.1 =>  (0x00007ffc9cbb0000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f6040d31000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f60410ff000)
[wongwf@asura vsyscall]$ ldd /bin/uname
        linux-vdso.so.1 =>  (0x00007ffc50f35000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fa0ae597000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fa0ae965000)
[wongwf@asura vsyscall]$ ldd /bin/uname
        linux-vdso.so.1 =>  (0x00007ffdff1ac000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f2982126000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f29824f4000)
```

Address randomized

Kernel-GLIBC

- Kernel provide the dynamic shared object to the loader
  - `vdso{32|64}.so` in `arch/x86/entry/vdso`
- Kernel detects loading of shared executable, will then provide the shared object in the process image
  - No file object
- Final setup done by glibc ELF startup code

What's inside vDSO?

- Sources are in arch/x86/entry/vdso
- Contains:
  - `__vdso_clock_gettime`
  - `__vdso_clock_getres`
  - `__vdso_getcpu`
  - `__vdso_gettimeofday`
  - `__vdso_time`

END