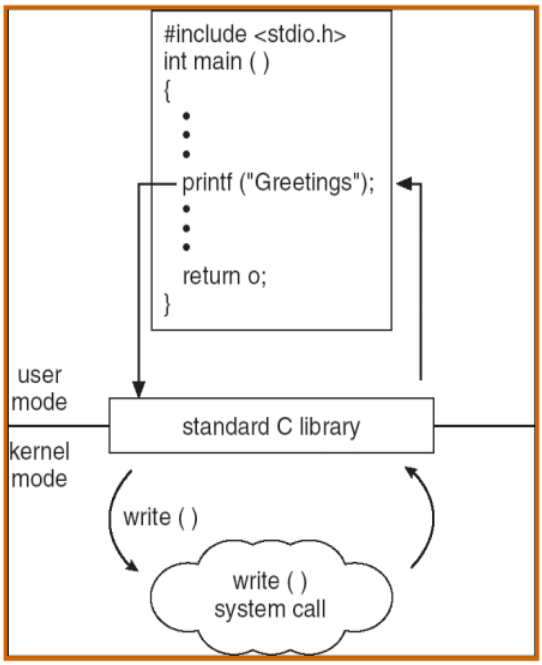
Topic: System calls vs System Call API and implementation of System Calls
Roll No. 2020201026

1. Introduction to System Calls [i][ii]

System call or a syscall is a method provided by the operating system to the application programs to communicate with the kernel. Processes in user space make use of system calls to interact with hardware devices such as disks, CPU, input devices and output devices. System calls act as a link between the two key access and security modes – user mode and kernel mode, for processing CPU commands. Until the request made by the system call has been served, the kernel takes control of the process which temporarily stops running, waiting for the execution of the system call to finish and to get back the control from the kernel. On carrying out the action required by the system call and on getting the control back, the execution of the program code is continued from the point where system call was made. This technique of system calls increases the system security as only the kernel can perform system critical actions that are blocked in user mode.

2. Introduction to System Call API [iii][iv][v]

Firstly, an Application Programming Interface (API) in general is a set of protocols, functions and routines that define methods of communication between different software programs and devices. The purpose of an API is to simplify programming by abstracting the underlying implementation and only exposing objects or actions the programmer needs. An API could issue several system calls or could also offer its services in the user mode directly (eg. Math functions).



A System Call API is a specific category of API that provides an interface through which user-level processes can make a system call to request services of the operating system. There exist several libraries of functions that provide such APIs to the programmers. Eg. some of the APIs defined by the standard C library (libc) refer to wrapper routines whose purpose is to issue system calls. If the programmer wants to allocate memory of a particular size, he can use the malloc() API implemented by the libc library which in turn invokes the required system calls to get the memory allocated.

The most common System Call API standards are :

- POSIX API : for POSIX-based systems (all versions of UNIX, LINUX, Mac OS X)
- Win32 API : for Microsoft Windows
- Java API : for Java Virtual Machine

3. Difference between System Call and System Call API [vi][vii][viii]

- The system call API is a function definition that specifies how to obtain a given service. Whereas the system call is an explicit request to the kernel for a service, made via a software interrupt.

- System calls belong to the kernel. While system call APIs are provided by several libraries that belong to the user mode.
- System calls are kernel level programs that provide a system service upon invocation, while system call APIs consists of functions which, upon being called by the user, invoke one or more required system calls. (As shown in the image above, printf() is a function provided by libc API library, which then calls the actual write() system call)
- For a programmer, the only things that matter are the API function's name, parameters and return code. However, a kernel developer needs to look after the system call implementation without worrying about the API libraries.
- Thus, for architectural visualization, we can say that the system call API is an intermediatory between the user programs and the actual system calls.

4. Classes of System Calls [ix][x]

1) **Process Control**: Create process, terminate process, load, execute, get or set process attributes, wait for an event, wait for some time, signal event, allocate or free memory.
2) **File Management**: Create or delete file, open or close file, read or write in the file, reposition the file, get or set file attributes.
3) **Device Management**: Request or release device, read or write to device buffer, reposition (eg. Disk), get or set device attributes, attach or detach devices logically.
4) **Information Maintenance**: Get or set time and date, get or set system data, get or set process, file or device attributes.
5) **Communication**: Create or delete communication connection, send or receive messages, transfer status information, attach or detach remove devices.
6) **Protection**: Get or set file or device permissions.

5. Implementation of System Calls

5.1. System Call Handler and Service Routines [xi]

When a user mode process invokes a system call, the CPU switches from user mode to kernel mode and jumps to an assembly language function known as the System Call Handler. The user mode process passes few parameters when invoking a system call among which the one called system call number is used to uniquely identify which system call is needed (Eg. EAX register used in Linux for this purpose). The kernel uses a system call dispatch table to associate each system call number with its corresponding service routine. The k'th entry in this table contains the service routine address of the system call having number k. All system calls return an integer value but the convention of these return values may be different from those for wrapper routines. In the kernel, positive or 0 value indicates a successful completion of the system call, whereas negative values denote the negation of the error code that is returned to the user program in errno variable which is set by the wrapper routine. The structure of the system call handler is similar to that of an exception handler. Tasks performed by the system call handler are as follows:

1) Save the contents of the registers in the kernel mode stack and switch to kernel mode from user mode. This operation is coded in Assembly language.
2) Handle the system call by invoking the corresponding function called as system call service routine. These routines are written in C language.

3) After execution of the system call service routine, load the registers with the contents stored in the kernel mode stack and switch CPU back to user node from kernel mode.

5.2. Issuing a System Call

User process in Linux can invoke system calls in two different ways:

- By executing INT$0x80 assembly language instruction
- By executing sysenter assembly language instruction

5.2.A. Using INT$0x80 Instruction [xii]

The traditional way of issuing a system call is by generating a software interrupt with the use of INT assembly language instruction. The vector 128 (0x80 in Hex) is associated with the kernel entry point. During kernel initialization, the trap_init() function is invoked that sets up the Interrupt Descriptor Table entry corresponding to vector 128 as: set_system_gate(0x80, &system_call);. This call loads the following values into gate descriptor fields:

- Segment Selector: The _ _KERNEL_CS Segment Selector of the kernel code segment.
- Offset: The pointer to the system_call( ) system call handler.
- Type: Set to 15. Indicates that the exception is a Trap and that the corresponding handler does not disable maskable interrupts.
- Descriptor Privilege Level: Set to 3. This allows processes in user mode to invoke the exception handler.

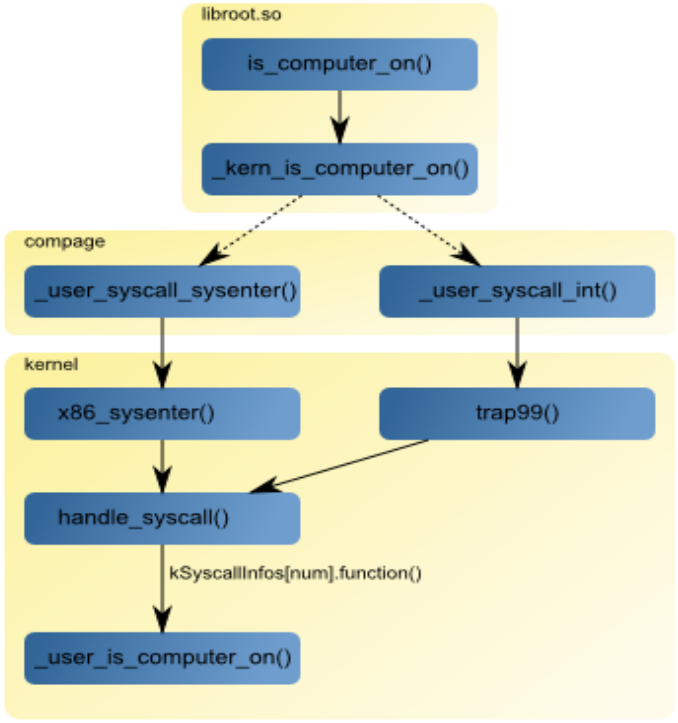
The CPU then switches to kernel mode and starts executing instructions of the system_call() function. It saves the system call number and all the CPU registers that may be used by exception handler on stack by using the SAVE_ALL macro. If the system call invocations of the executed program are being traced by a debugger, (i.e. if TIF_SYSCALL_ TRACE or TIF_SYSCALL_AUDIT flag in the thread_info structure is set), system_call() invokes the do_syscall_trace() function twice: once right before and once right after the execution of the system call service routine to allow the debugging process to collect information about it. The system call number passed by the user process is then checked for validity. If it is greater than the number of entries in the system call dispatch table, the system call handler terminates by storing –ENOSYS value in the stack location where EAX register has been saved. Then the process resumes its execution in user mode which finds the negative return code in the EAX register. If the system call number is valid, the specific service routine associated with the system call number contained in EAX is invoked by calculating its physical address. When the system call service routine terminates, the system_call() function gets its return code from EAX and stores it in the stack location where the User Mode value of the EAX register is saved. If there is no debugging, the contents of the registers saved on the Kernel Mode stack are restored and iret assembly language instruction is executed to resume the process in user mode.

5.2.B. Using sysenter instruction [xiii][xiv] [xv]

The INT assembly language instruction is inherently slow because it performs several consistency and security checks. The sysenter instruction, dubbed in Intel documentation as “Fast System Call,” provides a faster way to switch from User Mode to Kernel Mode. The sysenter assembly language instruction makes use of three special registers that must be loaded with the following information:

- SYSENTER_CS_MSR: The Segment Selector of the kernel code segment
- SYSENTER_EIP_MSR: The linear address of the kernel entry point
- SYSENTER_ESP_MSR: The kernel stack pointer

The SYSENTER instruction sets the following registers according to values specified by the operating system in certain model-specific registers. CS register is set to the value of (SYSENTER_CS_MSR), EIP register is set to the value of (SYSENTER_EIP_MSR), SS register is set to the sum of (8 plus the value in SYSENTER_CS_MSR) and ESP register is set to the value of (SYSENTER_ESP_MSR). Kernel also setups system call entry/exit points for user processes. Kernel creates a single page in the memory and attaches it to all processes' address space when they are loaded into memory. This page contains the actual implementation of the system call entry/exit mechanism. Kernel calls this page virtual dynamic shared object (vdso). User processes (C API library) call the __kernel_vsyscall to execute system calls. Address of __kernel_vsyscall is not fixed. Kernel passes this address to user processes using AT_SYSINFO elf parameter. AT_ elf parameters, also known as elf auxiliary vectors, are loaded on the process stack at the time of startup, along with the process arguments and the environment variables. After moving to this address, registers ECX, EDX and EBP are saved on the user stack and ESP is copied to EBP before executing sysenter. This EBP later helps kernel in restoring userland stack back. After executing sysenter instruction, processor starts execution at sysenter_entry. After saving the state, kernel validates the system call number stored in EAX. Finally appropriate system call is called using instruction: call *sys_call_table(,%eax,4). After system call is complete, the contents of the registers are copied back from the stack and the return value of sysenter is pushed to the stack. The SYSEXIT instruction is then executed and processor resumes execution in userland.



[i] https://www.ionos.com/digitalguide/server/know-how/what-are-system-calls/
[ii] Daniel P. Bovet , Marco Cesati - Understanding the Linux Kernel, Third Edition – O'Reilly Media (2005)
[iii] https://www.ionos.com/digitalguide/server/know-how/what-are-system-calls/
[iv] Daniel P. Bovet , Marco Cesati - Understanding the Linux Kernel, Third Edition – O'Reilly Media (2005)
[v] https://cw.fel.cvut.cz/old/_media/courses/ae3b33osd/osd-lec2-14.pdf
[vi] https://www.ionos.com/digitalguide/server/know-how/what-are-system-calls/
[vii] Daniel P. Bovet , Marco Cesati - Understanding the Linux Kernel, Third Edition – O'Reilly Media (2005)
[viii] https://cw.fel.cvut.cz/old/_media/courses/ae3b33osd/osd-lec2-14.pdf
[ix] https://en.wikipedia.org/wiki/System_call
[x] https://www.ionos.com/digitalguide/server/know-how/what-are-system-calls/
[xi] Daniel P. Bovet , Marco Cesati - Understanding the Linux Kernel, Third Edition – O'Reilly Media (2005)
[xii] Daniel P. Bovet , Marco Cesati - Understanding the Linux Kernel, Third Edition – O'Reilly Media (2005)
[xiii] Daniel P. Bovet , Marco Cesati - Understanding the Linux Kernel, Third Edition – O'Reilly Media (2005)
[xiv] http://articles.manugarg.com/systemcallinlinux2_6.html
[xv] https://www.haiku-os.org/documents/dev/system_calls/