**SYSTEM CALL**

**User vs Kernel Space :**

Before we talk about system call operations, we need to understand that most CPUs typically operate in at least two different modes: user mode and kernel mode (sometimes also referred to as supervisor mode). When running in user mode, the CPU can access only memory that is marked as being in user space; attempts to access memory in kernel space result in an exception. For example, a code

int x= 0;

scanf ( "%d", x) ;

will result in exception because the user program is trying to assign a value to address 0 which may be in kernel space. The actual code should be of course

scanf ( "%d", &x) ;

This ensures that user processes are not able to access the instructions and data structures of the kernel, or to perform operations that would adversely affect the operation of the system. When running in kernel mode, the CPU can access both user and kernel memory space.

**System Calls** A system call is a controlled entry point into the kernel, allowing a process to request the kernel perform some action on the process’s behalf. The kernel makes a range of services accessible to programs via the system call application programming interface (API). These services include, for example, creating a new process, performing I/O, and creating a pipe for interprocess communication. (The syscalls(2) manual page lists the Linux system calls and in our linux system the /usr/include/asm/unistd\_32.h) Before going into the details of how a system call works, some notable points are :

* A system call changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory.
* The set of system calls is fixed. Each system call is identified by a unique number. (This numbering scheme is not normally visible to programs, which identify system calls by name.) Each number is nothing but an array index to a data structure called System Call Table. Luckily, we don't have to worry about the numbers, we only have to know the system call name and parameters.
* Before kernel is starting to execute the system call, the user program may have a set of arguments that specify information to be transferred from user space to kernel space and vice versa.

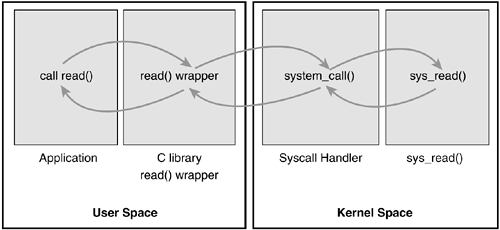
From a programming point of view, invoking a system call looks much like calling a C function. However, behind the scenes, many steps occur during the execution of a system call.

**syscall ( ) function - How does the system execute library calls in kernel space ?**

From a programming point of view, invoking a system call looks much like calling a C function. We know a function has a name and may have a bunch of input parameters. In the system, each system function name is mapped to a unique number. We will see how this works :

**IN USER SPACE:**

1. Because we cannot call the system function directly, a wrapper function is given to us. The application program makes a system call by invoking this wrapper function indirectly in the C library.



2. Because Kernel expects all arguments of the function in specific registers in the CPU, the wrapper function copies the arguments to these registers. These registers are mentioned below

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Architecture | arg1 | arg2 | arg3 | arg4 | arg5 | arg6 |
| i386 | ebx | ecx | edx | esi | edi | ebp |
| x86\_64 | rdi | rsi | rdx | r10 | r8 | r9 |
| x32 | rdi | rsi | rdx | r10 | r8 | r9 |

3. The wrapper function then copies the unique number onto a specific CPU register eax

4. Lastly, the wrapper function executes a trap machine instruction 0x80 which causes the processor to switch from user mode to kernel mode. New architectures use syscall. The program now in kernel space.

**IN KERNEL SPACE :**

5. Now in kernel space, in response to the trap to location 0x80, the kernel invokes its system\_call() routine to execute the function on behalf of the user program.

This routine :

a) Saves register values onto the kernel stack. That moves data to its internal memory.

b) VALIDATES THE FUNCTION CALL: Checks the validity of the system call number.

c) VALIDATES THE ARGUMENTS: If the system call service routine has any arguments, it first checks their validity.

d) Invokes the system call service routine mapped to the unique number.

e) Once the routine finishes, the service routine returns

d) Restores register values from the kernel stack and places the system call return value on the stack.

e) Returns to the wrapper function, simultaneously returning the processor to user mode.

6. If the return value of the system call service routine indicated an error, the wrapper function sets the global variable errno using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

Here is the simple C program :

int main ( )

{

printf ( " Hello World \n" ) ; gets converted into Syscall by the compiler

return 0 ;

}

The assembly code for the above C code is something like this:

.global \_start

.text

\_start:

# write(1, message, 13)

mov $4, %eax # system call 4 is write

mov $1, %ebx # file handle 1 is stdout

mov $message, %ecx # address of string to output

mov $13, %edx # number of bytes to write

int $0x80 # invoke operating system code

# exit(0)

mov $1, %eax # system call 1 is exit

xor %ebx, %ebx # we want return code 0

int $0x80 # invoke operating system code

message:

.ascii "Hello, World\n"