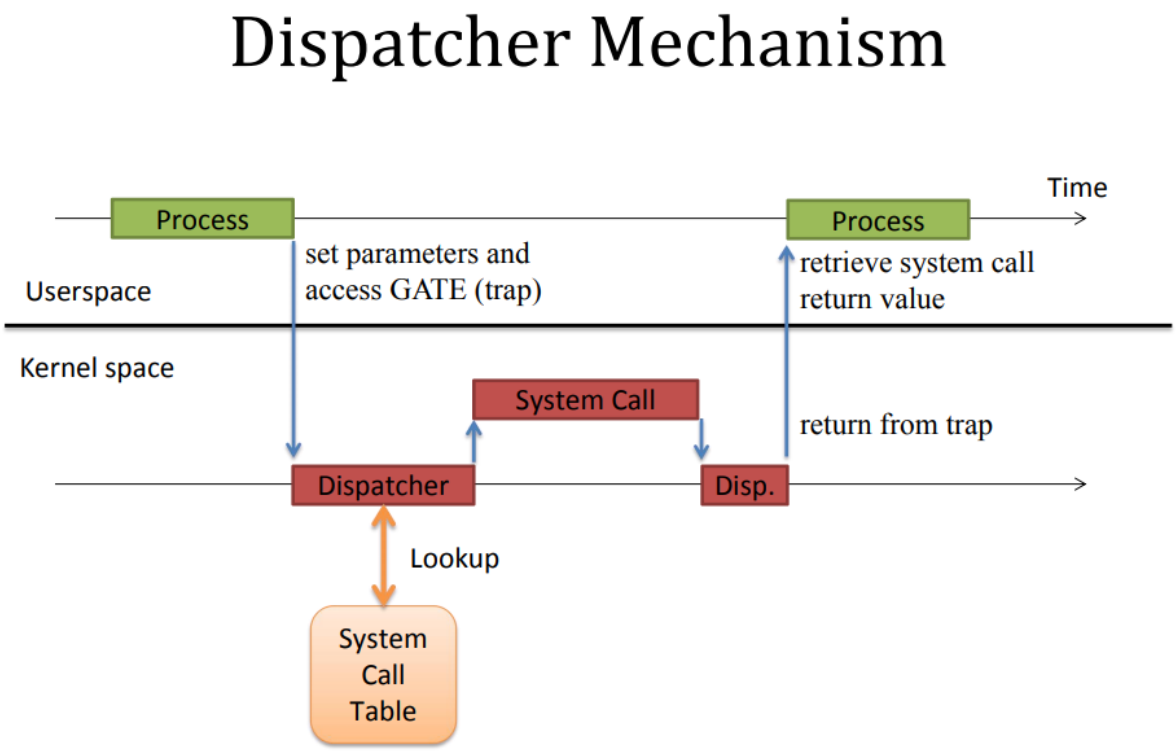
System Calls Management - (03-04-2020)

This is a fundamental subsystem of the kernel cause it is the entry point from userspace to kernel space. The goal is to hide all the hardware background forcing user applications to rely on a full API which are represented by system calls. The ring model allows us to differentiate between different context of execution and in order to transition from a less privileged like the user one to a more privileged like the kernel one, there is only one gate accessible via traps. But since the gate relies on traps mechanism and the number of total interrupts are bounded to the number of entries in the IDT (256), we have just a few number of these. Moreover a lot of entries are related to serve hardware interrupts, so the entries related to traps are smaller than 256. But the entire world of operating systems exploit a trick in order to implement more than 256 system calls: they use just one entry to call a dispatcher of an huge amount of system calls! In linux the entry index is 0x80. Globally there is the idt_table[NR_VECTORS] (NR_VECTORS is upper bounded by 256) in which at startup in function __init trap_init() each entry is initialized, at a certain point there is a call to a function called set_system_trap_gate(SYSCALL_VECTOR, &system_call). The main data structure is the SYSTEM CALL TABLE, in which each entry stores a pointer to kernel-level function activated by the dispatcher. The user space must communicate to the dispatcher via CPU registers some parameters in order to activate just a specific system call. A system call is identified by a unique integer which is also the offset in the system calls table. So the system call is activated via indirect call and the return value is stored in a register.



Notice that the return operation can't be done simply with a RET instruction cause the dispatcher was activated via traps mechanism, so the only way is to use an IRET instruction which in the very end restores also CS segment in order to give back control to user application. The int $0x80 trap mechanism in the early days was the only firmware facility that programmers could exploit in order to implements syscalls mechanism. It is very costly. During the time it changed thanks to new facilities that were implemented by Intel. We have to start know the trip around syscall mechanism but remember that since kernel wants to be backward compatible also this mechanism is supported nowadays.

First of all let's analyze the first trap mechanism. In order to trigger a trap programmers needed some C facilities. This functionality doesn't exist but it was thought to rely on MACROS definition in order to bind a syscall to a set of asm instruction which in the very end will trigger int $0x80. This fundamental operation is done by the UNISTD.H, if we not include that header we cannot invoke a system call in C. In that header there are also all the numerical codes associated to the syscalls, in this way it can prepare all the parameters to pass to the dispatcher. It is a common practice to make the bind within numerical code and syscall permanently for a specific architecture in order to not recompile entire programs if a change happens in a newer version of the kernel. But of course we can introduce new syscalls. Moreover there is a macro for each range of parameters accepted: since this scheme is working on a 32 bit systems, the number of parameters is between 0 and 6. So each macro which is bind to a syscall is also related to a macro that specifies the number of parameters that the syscall requires. The previous versions of the kernel maintained an header file with the definitions of each syscall number, nowadays it is generate at compile time thanks to a bash script which exploits a text file with a .tlb extension. Each definition is in the form __NR_#NameSyscall nr. The first numerical code is 1, cause the 0 is associated to a NISYSCALL which represents the identity syscall or the null one, if we call this function the kernel will immediately return control to the user application. It is a sort of placeholder for all the syscalls that are deprecated or reserved.

Macro for a 0-Parameters Syscall

```
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name)); \
__syscall_return(type,__res); \
}
```

Example syscall: `fork()`

Let's look to a macro for a syscall which requires 0 parameters. We have the definition of a _syscall0(type, name). This means that in the header we will find, for example for the FORK syscall that takes no parameters, a definition like #define fork() _syscall0(int, fork). The first parameters is the type of the return value, the second one is the name of the syscall. This line triggers the code generation relying on the macro in the above picture. Moreover, the _syscall0 definition will be translated into a function defined like type name(void). In the generated function we declare a res variable which will be the return value of the syscall, in fact it is mapped to the A register, and then we rely on the __asm__ volatile facility in which we explicitly call the int $0x80 instruction passing as the unique parameter the number of the syscall identified through its name (__NR_##name when ##is the concat operator and name the input parameter of the macro). Notice that the parameter is mapped again to the first register used, in this case A. At the end we

mangle the return value, stored in the A register, through another macro called __syscall_return(type, __res).

Return from a syscall

```
/* user-visible error numbers are in the range -1 - -124:
   see <asm-i386/errno.h> */

#define __syscall_return(type, res) \
do { \
        if ((unsigned long)(res) >= (unsigned long)(-125)) { \
                errno = -(res); \
                res = -1; \
        } \
        return (type) (res); \
} while (0)
```

Only if `res` in [–1, -124]

What's that?!

This macro is defined like as a do-while constructor. It is a do-while FALSE block. Why this? Cause some programmers doesn't use braces if an IF STATEMENT contains just one line. This is a problem cause if we don't use the do-while block the compiler will take on the first block of the code (in this case the first IF STATEMENT). In order to take atomically all the code we include it in a single statement of do-while false (which will be useless of course). After we check if the res value is in the range [-124, -1]. ERRNO is a global variable, a thread local one in a concurrent system, which tells us what is the fail cause of the syscall. It is a userspace variable so the kernel can't set it directly, but if the error is in that range it means that it is the ERRNO value. So what we have to do is to set in the userspace the correct value of ERRNO (which is the positive one of the error code returned by the kernel) and the set the return value in res to -1. In this way an user application can test the value in res and if it negative can check the corresponding ERRNO code to know about the reason of failure. If the kernel value is not in that range it will be directly returned (also if the syscall return a positive value). A lot of things like this are exposed by the standard library, cause the kernel doesn't want to deal with a lot of userspace related stuff. This macro is necessary cause the conventions are different: kernel returns a negative value which identifies the cause of the failure, instead user application requires a negative value and the ERRNO variable set accordingly.

The macro for a 1-Parameter Syscall is not so different from the one used for the 0-Parameter syscall one. We have a #define _syscall(type, name, type1, arg1) definition. The type1 and arg1 are resp. the type and the name of the first required parameter of the syscall. The remaining code is the same except for the fact that we have to pass one more parameter: it is passed into the B register. Why that? In this case we are explicitly break the 32 bit calling conventions cause the people that have written this piece of code are aware of the fact that the dispatcher (a SPECIFIC KERNEL FUNCTION) requires parameters in registers in order to be as fast as possible to serve a syscall. It is possible to do this because we are generating directly asm code. Moreover we use the B register cause it is a caller one: it means that since the syscall is translated into a function through the macro, it will be a task of the caller save the content of the caller registers, and so the asm code can be so small. It is a good practice to use a caller register in a function.

Macro for a 6-Parameters Syscall

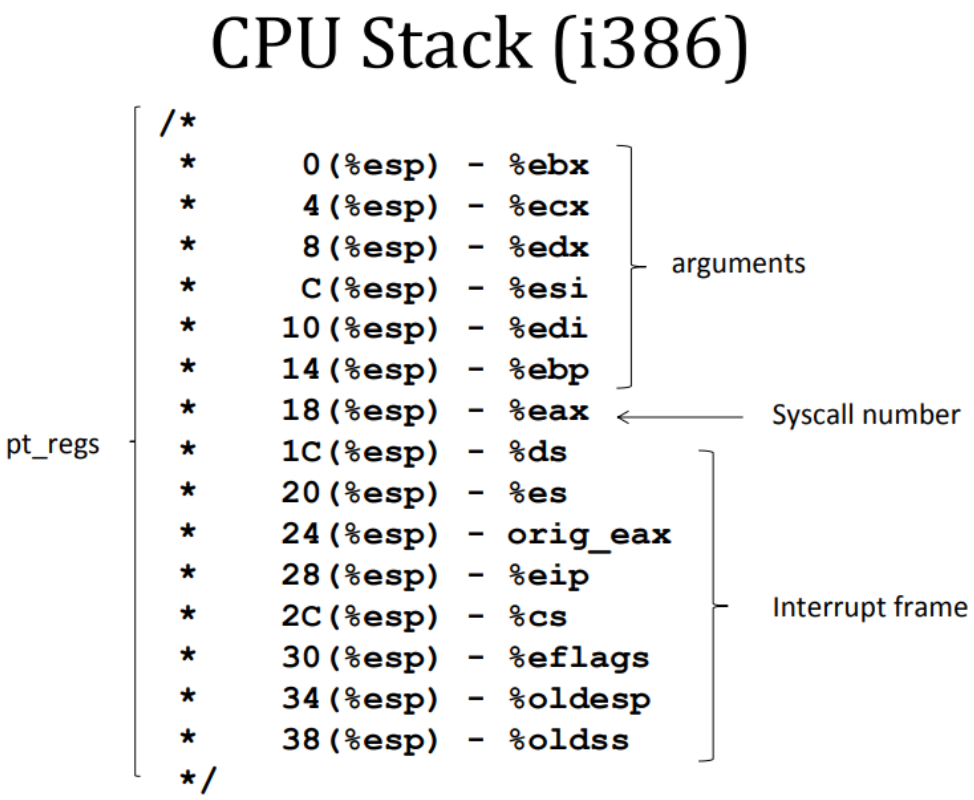
```
#define _syscall6(type,name,type1,arg1,type2,arg2,\
                  type3,arg3,type4,arg4,type5,arg5,type6,arg6) \
type name (type1 arg1,type2 arg2,type3 arg3,\
           type4 arg4,type5 arg5,type6 arg6) \
{ \
       long __res; \
       __asm__ volatile (
              "push %%ebp ; movl %%eax,%%ebp ;"\
              "movl %1,%%eax ; int $0x80 ; pop %%ebp" \
              : "=a" (__res) \
              : "i" (__NR_##name),"b" ((long)(arg1)),\
                "c" ((long)(arg2)),"d" ((long)(arg3)),\
                "S" ((long)(arg4)),"D" ((long)(arg5)), \
                "0" ((long)(arg6))
       ); \
       __syscall_return(type,__res); \
}
```

For a 6-Parameters Syscall things are more complicated. Let's have a look. Obv the definition is longer in order to specify all the types and names of the parameters. The problem here is the fact that on a 32 bit system we have just 8 general purpose registers and we want to use 6 of them to pass the parameters to the dispatcher. We have consumed all the caller save registers and we have to use ebp register which is a callee save one. The ebp is used to identify the function stack frame and if we clobber it we couldn't no longer to access stack variable of this function. Now we have to underline the fact that there is some code which is executed before our template, in this case our rules for mapping value into registers. So we cannot directly use ebp as the 6th register because it is a callee save one and it will be clobbered before saving it. So what we do is that we use in the input list the A register (a caller save one) to store the arg6, then we specify that the number of the syscall is an immediate (which index is 1 due to the order of the input list), then in the asm template (run after the movs operations to set registers content) we manually move the eax content into the ebp (remember that we first push ebp and eax contained the arg6), after we move the index 1 parameter (the immediate syscall number) into eax (as the dispatcher calling convention requires) and finally trigger the trap mechanism. After all we can restore the content previously saved of the ebp register.

Let's analyze the dispatcher's activity. Once we entered in kernel mode, the first thing that the dispatcher does is to take a complete snapshot of CPU registers. The trap activation must be completely transparent to the userspace application. The snapshot is saved in the system-level stack cause the dispatcher runs in R0 and thanks to the TSS the stack was already changed. Then the system call is invoked as a subroutine call (via call instruction) and it retrieves parameters from stack via the base pointer. Immediately after the stack change, on the system-level stack are saved the minimum amount of information (called interrupt frame) to manage the interrupt like the CS and the SS of the caller (in this case of ring 3) and obv the return address. These information are necessary in order to execute the IRET instruction at the end of the handling of an interrupt. Notice that the discussion about the magic behind the macros is related to a 32 bit systems which break the calling convention of passing parameters on the stack, instead the calling convention of the 63 bit systems is directly in according with this scheme, obv the registers used are in a different order, but the important thing to remember is that when we entered in the dispatcher routine in the CPU we have the parameters that we need. So the dispatcher can retrieve from them the arguments (optional) and the syscall number (mandatory obv) and put

them on the stack, in a certain order such that the RSP points to the first argument until the last, then (the farthest but the first to be pushed) is the number of the syscall (E/RAX). Then we can use the syscall number in order to call the right function associated to the syscall that we want to activate. The syscalls are declared with the attribute "asmlinkage" cause they must be aware of the way the dispatcher pass the parameters, in this case it pushes them onto the stack.

This sort of preamble which the dispatcher is in charge to perform before calling the syscall is architecture dependent. In this way all the syscall mechanism will be the same over all the supported architectures. In case something must be changed, the only part of the code that it is needed to be touched is the "left" part, I mean the from the dispatcher to the libc. The stack is interpreted by the syscall thanks to a fundamental structure called PT_REGS, which is a pointer to the system-level stack (points to the data prepared by the dispatcher, to the first element arg1).



For the 32 bit system, the CPU stack in terms of pt_regs is this one. First of all we have arguments, then the syscall number, finally the interrupt frame saved in order to restore the previous user context. Notice that the interrupt frame is put there by the firmware! So the dispatcher must put onto the stack only the last two set of values.

Syscall Dispatcher (i386)

```
ENTRY(system_call)
        pushl %eax  # syscall no.
        SAVE_ALL
        GET_CURRENT(%ebx)
        testb $0x02, tsk_ptrace(%ebx) # PT_TRACESYS
        jne tracesys
        cmpl $(NR_syscalls),%eax
        jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax, EAX(%esp)    # save the return value
ENTRY(ret_from_sys_call)
        cli             # need_resched and signals atomic test
        cmpl $0, need_resched(%ebx)
        jne reschedule
        cmpl $0, sigpending(%ebx)
        jne signal_return
restore_all:
        RESTORE_ALL
```

This is the code of the dispatcher function in the 32 bit systems. It is quite simple. This is the entry point of the dispatcher stored in the corresponding gate descriptor, the first code activated when we transition at R0. The first instruction is a push of the syscall number, then we have a macro to push all the remaining arguments. After calling into the system table we have to check if the number passed by the userspace application is a valid one. We don't care if the syscall clobbers registers because all its arguments are as well stored in the stack. When the syscall returns, we save the content of the EAX register in the actual EAX of the PT_REGS struct and the we rely on a macro that RESTORE_ALL registers (EAX will store the return value, no longer the syscall number). But before returning, we check if there is the need of a reschedule of a more important task or if there is need of activating some signal handler which is in pending. This is the reason why we save onto the stack (in the PT_REGS struct) the return value of the syscall, because the probability of clobbering EAX content if we jump in one of the two function described before, is up to 100%.

But the approach behind the generation of a trap is very costly: first of all we need to access the IDTR register to find the base pointer of the IDT, then we apply the offset specified by the instruction (0x80 for syscall) an we find an entry in which there are a selector and an offset, the selector (obv) select an entry (gate for syscall) in the GDT whose base address is retrieved accessing the GDTR register, then we find the base offset in main memory of the kernel code and thanks to the offset of the IDT we can activate our routine (dispatcher). A looooooooooong trip for any syscall only to reach the very same address of the dispatcher. Can we do better?

So vendors started to propose new asm instructions. Initially AMD proposed syscall/sysret in order to trigger a trap to execute a generic system call, sysenter/sysexit have been introduced (for 32 bit systems). So again there is just one single entry point for the firmware and again this requires a system call number.

Let's focus on sysenter/sysexit. They doesn't require to rely on a software trap in order to activate the dispatcher. In fact are called Fast syscall Path. They rely on MSR registers which in a certain way precache the required information to call the dispatcher previously retrieved by the IDTR and the GDTR and so on. In fact these registers store the CS and EIP to run at ring 0 and the pointer to the dispatcher (stored resp. SYSENTER_CS_MSR and SYSENTER_CS_MSR). These information are

set at startup by the kernel cause they will not change during the kernel life. The other piece of information necessary are the SS segment and the stack pointer ESP in order to use the system-level stack after the context switch. Moreover the sysenter instruction will first of all save the content of CS, EIP, SS, ESP related to the caller user application cause at the end of the syscall, sysexit will restore them correctly. The EIP of the user will be saved into the EDX register and ESP into the ECX register. This will not create a problem, cause as we said the dispatcher will create a complete snapshot of the CPU registers. To use information related to the MSR register, we rely on definitions and to write on them we use WRMSR asm instruction. At startup the CS is set to the __KERNEL_CS obv, the instruction pointer EIP is set to the sysenter_entry (dispatcher) and the stack pointer ESP is set to the R0 stack stored in the TSS.

For 64 bit systems things are quite different. It is used two different asm instructions called SYSCALL/SYSRET. Again based on MSRs. We rely on just one modern specific register called STAR/LSTAR. Why two names? Cause this allow us to write in different portion of the same register. When we use STAR we can write Kernel CS and the User CS, we could write also an address as the entry point of the dispatcher, but since the STAR is specified as 32 bit, we can only write a 32 bit address. So the LSTAR allow us to specify the entry point as a 64 bit address, but we can't use it to specify also KCS and UCS. So at startup kernel wrmsrl the KCS and UCS through the STAR and the entry point through LSTAR.

Recap: syscall() libc function, depending on the architecture, will use the new asm instructions in order to avoid the MACROS mechanism previously seen. At 32 bit system it uses SYSENTER and SYSEXIT, passing the syscall number into EAX and other eventual parameters into other registers accordingly. Then SYSENTER will look to the information stored into MSRs and after the context switch it will call the dispatcher. SYSEXIT will restore the user application context. On 64 bit the approach is the same but we now rely on SYSCALL and SYSRET instructions. The syscall number is always passed into RAX but now the MSR is one called LSTAR/STAR. The procedure is the same but the order convention for parameters in the register is different due to the increased number of available registers as follow. So since the calling conventions on 64 bit system is already set to parameters in register, we can rely on that order with a little change: RAX contains syscall number, from arg0 to arg5 are used the same register for the userspace application, BUT for the third instead of RCX it is used R10. Why this? Cause the RCX is used by the firmware in order to restore the instruction pointer (return address) for the user application. Moreover r11 is used to save the EFLAGS register of the user context. Notice that every time we execute the SYSCALL instruction, we enter into the dispatcher with interrupts disabled and this instruction is only callable from user space in order to enter in kernel space.

Now the complete scenario is better than the previous, cause we are use asm instructions which are dedicated to the syscall activation and so very less costly that the first implemented scheme and moreover the userspace code is simpler thanks to the syscall() library function that avoid the MACROS based scheme that was very complex. But actually the maintainers of the library have to manage code that is related to the kernel stuff. The main idea is to give responsibilities of all the dispatcher activation to kernel level programmers. But this is a problem cause the syscall library function runs at ring 0 and this is cause the syscalls must be activated only from userspace.

This problem was solved by kernel programmers thanks to the creation of Virtual Dynamic Shared Object (vDSO). This is basically a shared library maintained by kernel developers and of course it is part of the kernel source tree. The trick is that every time kernel spawns a new process, before to give control to him, it loads and maps in the virtual address space of the process this shared library in order to let the process use it. Thanks to that this code can be run at ring 3 and contemporaneously be maintained by kernel developers. So the point is: for some syscalls, there is need to run its code at ring 0? For example, the gettimeofday() syscall simply read the current kernel time, it is considered safe and so its code was moved into the vDSO in order to make things faster without the context switch. So vDSO contains safe syscalls which can

be execute at ring 3 and obv the portion of the code which allow user to transition in kernel mode. The syscall library function will rely on the vDSO in order to activate the dispatcher. The vDSO is loaded into a different location (some kind of ASLR) for each process, this is why we want to prevent attacker to use kernel code gadgets. The random address of the vDSO is loaded by the kernel into the ELF header table of the process. In this way thanks to GETAUXVAL function the syscall() library function can easily retrieve it.

vDSO Entry Point

```
__kernel_vsyscall:
        push %ecx
        push %edx
        push %ebp
        movl %esp,%ebp
        sysenter
        nop
        /* 14: System call restart point is here! */
        int $0x80
        /* 16: System call normal return point is here! */
        pop %ebp
        pop %edx
        pop %ecx
        ret
```

This is the activation path for the syscall (32 bit code). First of all we push and so save ECX and EDX cause SYSENTER will use them as we said. Then we create a function frame for the current execution context and we execute the SYSENTER. If we run this code on a machine that doesn't support the sysenter instruction, it will be generated a General Protection Fault which will generate a trap. The SO will understand that this fault is given by the no opcode mapping for the sysenter instruction and it will give back control to the next instruction of the function, which is the traditional int $0x80 (backward compatible also for the fast syscall path activation). At the end we restore registers content and return to our caller -> syscall() library function. The vDSO bypasses segmentation! So it is more efficient. We have benefit up to 75% in performance. Also we enhanced security thanks to randomization. The OS maintaines two different version of the vDSO: 32 bit and 64 bit.

The first version of the syscall table was written directly in assembly and store in a file called entry.S. From Kernel 2.6 32 bit and 64 bit versions were put together in the same folder and to each one was associated an assembly file called syscall_table32/64.S. In more recent versions there is a file .tbl for each version (32/64 bit) in which it is specified the way according to the table must be created by a shell script at compile time. In each entry it is stored a pointer to the architecture independent entry point of a kernel-level system call which is called traditionally with the prefix "sys_#nameofthesyscall".

In the syscall_64.tbl file (a text file) we can find the declaration of the syscalls for 64 bit architecture: for example for the read we have a line "0 common read __x64_sys_read". 0 is the syscall number, common is just an attribute which specifies the visibility of this function to other modules, then we find the name of the syscall and finally the actual entry point.

Then in the syscalls.h header file there is the definition of this function defined with asmlinkage attribute of course and in which are specified all the input parameters and the type of the return value. The name is as we said sys_read but it is just the architecture independent entry point, so the asmlinkage is used to clean up the different calling conventions among the different architectures. Somewhere this function will be declared.

The read syscall is part of the file system and so in the read_write.c file there is its implementation. But here we don't have directly the sys_read symbol (what? again? how much macros have been defined by Torvalds? :( ). Kernel generates sys_read symbol relying on an additional macro called SYSCALL_DEFINE3. The 3 specifies how much parameters the syscall takes (similarly to MACROS used in the library for 32 bit system in order to trigger the trap mechanism). This macros takes the name of the syscall ("read"), the type of the argument followed by its name (ex. unsigned int, fd). This macros boils down in a call to a kernel function called ksys_read(fd, buf, count). Why this? So, SYSCALL_DEFINE3 is generated by a general macro called SYSCALL_DEFINEx which take as the first parameter the number of parameters x. The SYSCALL_DEFINEx macro will set some metadata about the syscall thanks to SYSCALL_METADATA macro and will boil down calling __SYSCALL_DEFINEx macro very complex.

C Syscall Entry Points (4.20)

```
asmlinkage long sys_read(unsigned int fd, char __user * buf, size_t count)
__attribute__((alias(__stringify(SyS_read))));
```

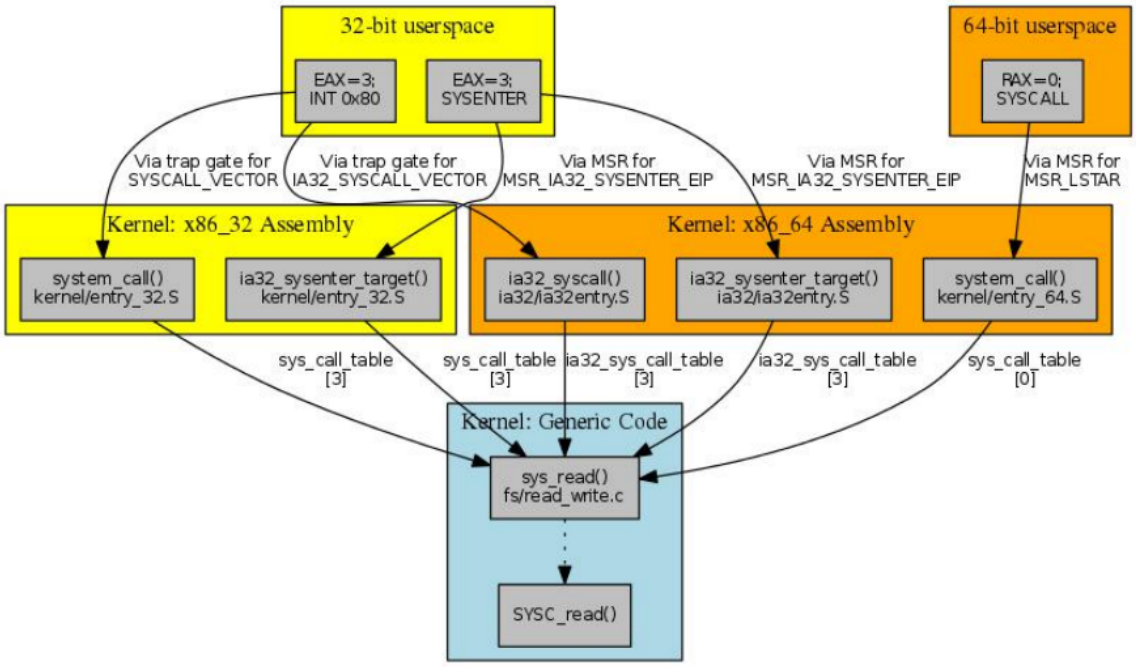
for security reasons (sign extension)

```
asmlinkage long SyS_read(long int fd, long int buf, long int count)
{
    long ret = SYSC_read((unsigned int) fd, (char __user *) buf, (size_t) count);
    asmlinkage_protect(3, ret, fd, buf, count);
    return ret;
}

static inline long SYSC_read(unsigned int fd, char __user * buf, size_t count)
{
    return ksys_read(fd, buf, count);
}
```

This is the actual expansion of the __SYSCALL_DEFINEx macro in the case of the read syscall. The first line is the declaration of our syscall with the sys_read symbol associated. It is just an alias of a SyS_read function and the compiler will substitute the sys_read symbol with the implementation of the alias. Why this? Just for security reason! Look at the parameters definitions, they are different. This is done cause since the kernel can be compiled at 64 bit, it must run also code compiled for 32 bit architecture. In this way an unsigned int (first parameter) will be sign extended to 64 bit while executing on a 64 bit kernel. Then the alias will call the last wrapper passing the right casted parameters to the real implementation of the read syscall. Moreover security is enhanced thanks to the asmlinkage_protect directive which ensures the fact that only the 3 valid parameters of the read syscall (or what else syscall) are used. Notice that in the .tbl file there was the __x64_sys_read symbol which is a valid entry point in this complex scheme, it points directly to the SyS_read function cause, being used in a 64 bit system, it doesn't need the sign extension protection so far discussed.

The Final Picture



Look that and cry. It is the final picture of syscall paths activation.