Chapter 10. System Calls

Operating systems offer processes running in User Mode a set of interfaces to interact with hardware devices such as the CPU, disks, and printers. Putting an extra layer between the application and the hardware has several advantages. First, it makes programming easier by freeing users from studying low-level programming characteristics of hardware devices. Second, it greatly increases system security, because the kernel can check the accuracy of the request at the interface level before attempting to satisfy it. Last but not least, these interfaces make programs more portable, because they can be compiled and executed correctly on every kernel that offers the same set of interfaces.

Unix systems implement most interfaces between User Mode processes and hardware devices by means of *system calls* issued to the kernel. This chapter examines in detail how Linux implements system calls that User Mode programs issue to the kernel.

10.1. POSIX APIs and System Calls

Let's start by stressing the difference between an application programmer interface (API) and a system call. The former is a function definition that specifies how to obtain a given service, while the latter is an explicit request to the kernel made via a software interrupt.

Unix systems include several libraries of functions that provide APIs to programmers. Some of the APIs defined by the *libc* standard C library refer to *wrapper routines* (routines whose only purpose is to issue a system call). Usually, each system call has a corresponding wrapper routine, which defines the API that application programs should employ.

The converse is not true, by the wayan API does not necessarily correspond to a specific system call. First of all, the API could offer its services directly in User Mode. (For something abstract such as math functions, there may be no reason to make system calls.) Second, a single API function could make several system calls. Moreover, several API functions could make the same system call, but wrap extra functionality around it. For instance, in Linux, the `malloc( )`, `calloc( )`, and `free( )` APIs are implemented in the *libc* library. The code in this library keeps track of the allocation and deallocation requests and uses the `brk( )` system call to enlarge or shrink the process heap (see the section "Managing the Heap" in Chapter 9).

The POSIX standard refers to APIs and not to system calls. A system can be certified as POSIX-compliant if it offers the proper set of APIs to the application programs, no matter how the corresponding functions are implemented. As a matter of fact, several non-Unix systems have been certified as POSIX-compliant, because they offer all traditional Unix services in User Mode libraries.

From the programmer's point of view, the distinction between an API and a system call is irrelevantthe only things that matter are the function name, the parameter types, and the meaning of the return code. From the kernel designer's point of view,

however, the distinction does matter because system calls belong to the kernel, while User Mode libraries don't.

Most wrapper routines return an integer value, whose meaning depends on the corresponding system call. A return value of -1 usually indicates that the kernel was unable to satisfy the process request. A failure in the system call handler may be caused by invalid parameters, a lack of available resources, hardware problems, and so on. The specific error code is contained in the `errno` variable, which is defined in the *libc* library.

Each error code is defined as a macro constant, which yields a corresponding positive integer value. The POSIX standard specifies the macro names of several error codes. In Linux, on 80 x 86 systems, these macros are defined in the header file *include/asm-i386/errno.h*. To allow portability of C programs among Unix systems, the *include/asm-i386/errno.h* header file is included, in turn, in the standard */usr/include/errno.h* C library header file. Other systems have their own specialized subdirectories of header files.

10.2. System Call Handler and Service Routines

When a User Mode process invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function. As we will see in the next section, in the 80 x 86 architecture a Linux system call can be invoked in two different ways. The net result of both methods, however, is a jump to an assembly language function called the *system call handler*.

Because the kernel implements many different system calls, the User Mode process must pass a parameter called the *system call number* to identify the required system call; the `eax` register is used by Linux for this purpose. As we'll see in the section "Parameter Passing" later in this chapter, additional parameters are usually passed when invoking a system call.

All system calls return an integer value. The conventions for these return values are different from those for wrapper routines. In the kernel, positive or 0 values denote a successful termination of the system call, while negative values denote an error condition. In the latter case, the value is the negation of the error code that must be returned to the application program in the `errno` variable. The `errno` variable is not set or used by the kernel. Instead, the wrapper routines handle the task of setting this variable after a return from a system call.

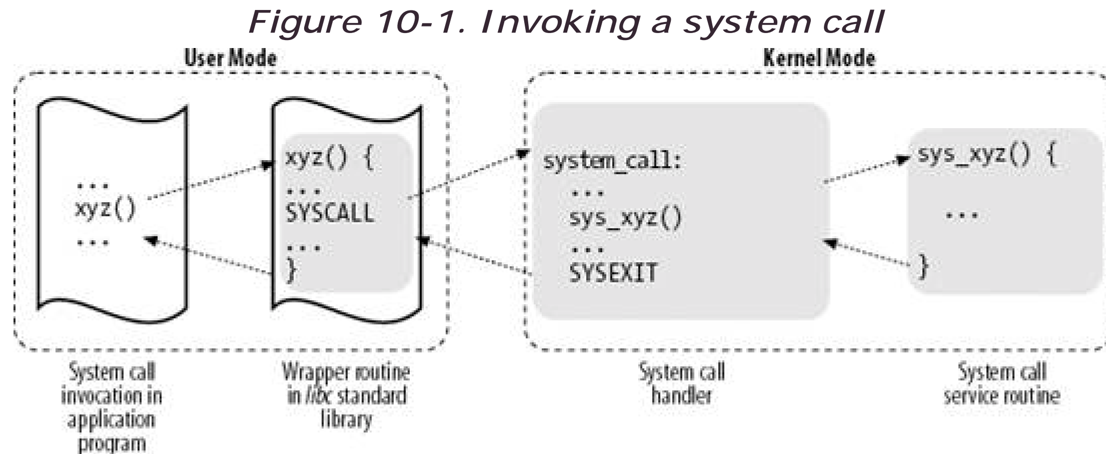
The system call handler, which has a structure similar to that of the other exception handlers, performs the following operations:

- Saves the contents of most registers in the Kernel Mode stack (this operation is common to all system calls and is coded in assembly language).
- Handles the system call by invoking a corresponding C function called the *system call service routine*.
- Exits from the handler: the registers are loaded with the values saved in the Kernel Mode stack, and the CPU is switched back from Kernel Mode to User Mode (this operation is common to all system calls and is coded in assembly language).

The name of the service routine associated with the `xyz( )` system call is usually `sys_xyz( )`; there are, however, a few exceptions to this rule.

Figure 10-1 illustrates the relationships between the application program that invokes a system call, the corresponding wrapper routine, the system call handler, and the system call service routine. The arrows denote the execution flow between the functions. The terms "`SYSCALL`" and "`SYSEXIT`" are placeholders for the actual assembly language instructions that switch the CPU, respectively, from User Mode to Kernel Mode and from Kernel Mode to User Mode.

Figure 10-1. Invoking a system call



To associate each system call number with its corresponding service routine, the kernel uses a *system call dispatch table*, which is stored in the `sys_call_table` array and has `NR_syscalls` entries (289 in the Linux 2.6.11 kernel). The nth entry contains the service routine address of the system call having number *n*.

The `NR_syscalls` macro is just a static limit on the maximum number of implementable system calls; it does not indicate the number of system calls actually implemented. Indeed, each entry of the dispatch table may contain the address of the `sys_ni_syscall( )` function, which is the service routine of the "nonimplemented" system calls; it just returns the error code `-ENOSYS`.

10.3. Entering and Exiting a System Call

Native applications[*] can invoke a system call in two different ways:

[*] As we will see in the section "Execution Domains" in Chapter 20, Linux can execute programs compiled for "foreign" operating systems. Therefore, the kernel offers a compatibility mode to enter a system call: User Mode processes executing iBCS and Solaris /x86 programs can enter the kernel by jumping into suitable call gates included in the default Local Descriptor Table (see the section "The Linux LDTs" in Chapter 2).

- By executing the `int $0x80` assembly language instruction; in older versions of the Linux kernel, this was the only way to switch from User Mode to Kernel Mode.

- By executing the sysenter assembly language instruction, introduced in the Intel Pentium II microprocessors; this instruction is now supported by the Linux 2.6 kernel.

Similarly, the kernel can exit from a system callthus switching the CPU back to User Modein two ways:

- By executing the iret assembly language instruction.
- By executing the sysexit assembly language instruction, which was introduced in the Intel Pentium II microprocessors together with the sysenter instruction.

However, supporting two different ways to enter the kernel is not as simple as it might look, because:

- The kernel must support both older libraries that only use the int $0x80 instruction and more recent ones that also use the sysenter instruction.
- A standard library that makes use of the sysenter instruction must be able to cope with older kernels that support only the int $0x80 instruction.
- The kernel and the standard library must be able to run both on older processors that do not include the sysenter instruction and on more recent ones that include it.

We will see in the section "Issuing a System Call via the sysenter Instruction" later in this chapter how the Linux kernel solves these compatibility problems.

10.3.1. Issuing a System Call via the int $0x80 Instruction

The "traditional" way to invoke a system call makes use of the int assembly language instruction, which was discussed in the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4.

The vector 128in hexadecimal, 0x80is associated with the kernel entry point. The trap_init( ) function, invoked during kernel initialization, sets up the Interrupt Descriptor Table entry corresponding to vector 128 as follows:

```
set_system_gate(0x80, &system_call);
```

The call loads the following values into the gate descriptor fields (see the section "Interrupt, Trap, and System Gates" in Chapter 4):

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.

DPL (Descriptor Privilege Level)

Set to 3. This allows processes in User Mode to invoke the exception handler (see the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4).

Therefore, when a User Mode process issues an int $0x80 instruction, the CPU switches into Kernel Mode and starts executing instructions from the system_call address.

10.3.1.1. The system_call( ) function

The system_call( ) function starts by saving the system call number and all the CPU registers that may be used by the exception handler on the stackexcept for eflags, cs, eip, ss, and esp, which have already been saved automatically by the control unit (see the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4). The SAVE_ALL macro, which was already discussed in the section "I/O Interrupt Handling" in Chapter 4, also loads the Segment Selector of the kernel data segment in ds and es:

```
system_call:
  pushl %eax
  SAVE_ALL
  movl $0xffffe000, %ebx /* or 0xfffff000 for 4-KB stacks */
  andl %esp, %ebx
```

The function then stores the address of the thread_info data structure of the current process in ebx (see the section "Identifying a Process" in Chapter 3). This is done by taking the value of the kernel stack pointer and rounding it up to a multiple of 4 or 8 KB (see the section "Identifying a Process" in Chapter 3).

Next, the system_call( ) function checks whether either one of the TIF_SYSCALL_TRACE and TIF_SYSCALL_AUDIT flags included in the flags field of the thread_info structure is setthat is, whether the system call invocations of the executed program are being traced by a debugger. If this is the case, system_call( ) invokes the do_syscall_trace( ) function twice: once right before and once right

after the execution of the system call service routine (as described later). This function stops current and thus allows the debugging process to collect information about it.

A validity check is then performed on the system call number passed by the User Mode process. If it is greater than or equal to the number of entries in the system call dispatch table, the system call handler terminates:

```
    cmpl $NR_syscalls, %eax
    jb nobadsys
    movl $(-ENOSYS), 24(%esp)
    jmp resume_userspace
  nobadsys:
```

If the system call number is not valid, the function stores the -ENOSYS value in the stack location where the eax register has been savedthat is, at offset 24 from the current stack top. It then jumps to resume_userspace (see below). In this way, when the process resumes its execution in User Mode, it will find a negative return code in eax.

Finally, the specific service routine associated with the system call number contained in eax is invoked:

```
    call *sys_call_table(0, %eax, 4)
```

Because each entry in the dispatch table is 4 bytes long, the kernel finds the address of the service routine to be invoked by multiplying the system call number by 4, adding the initial address of the sys_call_table dispatch table, and extracting a pointer to the service routine from that slot in the table.

10.3.1.2. Exiting from the system call

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:

```
    movl %eax, 24(%esp)
```

Thus, the User Mode process will find the return code of the system call in the eax register.

Then, the system_call( ) function disables the local interrupts and checks the flags in the thread_info structure of current:

```
    cli
    movl 8(%ebp), %ecx
    testw $0xffff, %cx
```

```
je restore_all
```

The `flags` field is at offset 8 in the `tHRead_info` structure; the mask `0xffff` selects the bits corresponding to all flags listed in Table 4-15 except `TIF_POLLING_NRFLAG`. If none of these flags is set, the function jumps to the `restore_all` label: as described in the section "Returning from Interrupts and Exceptions" in Chapter 4, this code restores the contents of the registers saved on the Kernel Mode stack and executes an `iret` assembly language instruction to resume the User Mode process. (You might refer to the flow diagram in Figure 4-6.)

If any of the flags is set, then there is some work to be done before returning to User Mode. If the `TIF_SYSCALL_TRACE` flag is set, the `system_call( )` function invokes for the second time the `do_syscall_trace( )` function, then jumps to the `resume_userspace` label. Otherwise, if the `TIF_SYSCALL_TRACE` flag is not set, the function jumps to the `work_pending` label.

As explained in the section "Returning from Interrupts and Exceptions" in Chapter 4, that code at the `resume_userspace` and `work_pending` labels checks for rescheduling requests, virtual-8086 mode, pending signals, and single stepping; then eventually a jump is done to the `restore_all` label to resume the execution of the User Mode process.

10.3.2. Issuing a System Call via the sysenter Instruction

The `int` assembly language instruction is inherently slow because it performs several consistency and security checks. (The instruction is described in detail in the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4.)

The `sysenter` instruction, dubbed in Intel documentation as "Fast System Call," provides a faster way to switch from User Mode to Kernel Mode.

10.3.2.1. The sysenter instruction

The `sysenter` assembly language instruction makes use of three special registers that must be loaded with the following information:[*]

[*] "MSR" is an acronym for "Model-Specific Register" and denotes a register that is present only in some models of 80 x 86 microprocessors.

`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

When the sysenter instruction is executed, the CPU control unit:

1. Copies the content of SYSENTER_CS_MSR into cs.
2. Copies the content of SYSENTER_EIP_MSR into eip.
3. Copies the content of SYSENTER_ESP_MSR into esp.
4. Adds 8 to the value of SYSENTER_CS_MSR, and loads this value into ss.

Therefore, the CPU switches to Kernel Mode and starts executing the first instruction of the kernel entry point. As we have seen in the section "The Linux GDT" in Chapter 2, the kernel stack segment coincides with the kernel data segment, and the corresponding descriptor follows the descriptor of the kernel code segment in the Global Descriptor Table; therefore, step 4 loads the proper Segment Selector in the ss register.

The three model-specific registers are initialized by the enable_sep_cpu( ) function, which is executed once by every CPU in the system during the initialization of the kernel. The function performs the following steps:

1. Writes the Segment Selector of the kernel code (_ _KERNEL_CS) in the SYSENTER_CS_MSR register.
2. Writes in the SYSENTER_CS_EIP register the linear address of the sysenter_entry( ) function described below.
3. Computes the linear address of the end of the local TSS, and writes this value in the SYSENTER_CS_ESP register.[*]

[*] The encoding of the local TSS address written in SYSENTER_ESP_MSR is due to the fact that the register should point to a real stack, which grows towards lower address. In practice, initializing the register with any value would work, provided that it is possible to get the address of the local TSS from such a value.

The setting of the SYSENTER_CS_ESP register deserves some comments. When a system call starts, the kernel stack is empty, thus the esp register should point to the end of the 4- or 8-KB memory area that includes the kernel stack and the descriptor of the current process (see Figure 3-2). The User Mode wrapper routine cannot properly set this register, because it does not know the address of this memory area; on the other hand, the value of the register must be set before switching to Kernel Mode. Therefore, the kernel initializes the register so as to encode the address of the Task State Segment of the local CPU. As we have described in step 3 of the _ _switch_to( ) function (see the section "Performing the Process Switch" in Chapter 3), at every process switch the kernel saves the kernel stack pointer of the current process in the esp0 field of the local TSS. Thus, the system call handler reads the esp register, computes the address of the esp0 field of the local TSS, and loads into the same esp register the proper kernel stack pointer.

10.3.2.2. The vsyscall page

A wrapper function in the *libc* standard library can make use of the `sysenter` instruction only if both the CPU and the Linux kernel support it.

This compatibility problem calls for a quite sophisticated solution. Essentially, in the initialization phase the `sysenter_setup( )` function builds a page frame called *vsyscall page* containing a small ELF shared object (i.e., a tiny ELF dynamic library). When a process issues an `execve( )` system call to start executing an ELF program, the code in the vsyscall page is dynamically linked to the process address space (see the section "The exec Functions" in Chapter 20). The code in the vsyscall page makes use of the best available instruction to issue a system call.

The `sysenter_setup( )` function allocates a new page frame for the vsyscall page and associates its physical address with the `FIX_VSYSCALL` fix-mapped linear address (see the section "Fix-Mapped Linear Addresses" in Chapter 2). Then, the function copies in the page either one of two predefined ELF shared objects:

- If the CPU does not support `sysenter`, the function builds a vsyscall page that includes the code:

```
    _ _kernel_vsyscall:
      int $0x80
      ret
```

- Otherwise, if the CPU does support `sysenter`, the function builds a vsyscall page that includes the code:

```
    _ _kernel_vsyscall:
      pushl %ecx
      pushl %edx
      pushl %ebp
      movl %esp, %ebp
      sysenter
```

When a wrapper routine in the standard library must invoke a system call, it calls the `_ _kernel_vsyscall( )` function, whatever it may be.

A final compatibility problem is due to old versions of the Linux kernel that do not support the `sysenter` instruction; in this case, of course, the kernel does not build the vsyscall page and the `_ _kernel_vsyscall( )` function is not linked to the address space of the User Mode processes. When recent standard libraries recognize this fact, they simply execute the `int $0x80` instruction to invoke the system calls.

10.3.2.3. Entering the system call

The sequence of steps performed when a system call is issued via the `sysenter` instruction is the following:

1. The wrapper routine in the standard library loads the system call number into the `eax` register and calls the `_ _kernel_vsyscall( )` function.
2. The `_ _kernel_vsyscall( )` function saves on the User Mode stack the contents of `ebp`, `edx`, and `ecx` (these registers are going to be used by the system call handler), copies the user stack pointer in `ebp`, then executes the `sysenter` instruction.
3. The CPU switches from User Mode to Kernel Mode, and the kernel starts executing the `sysenter_entry( )` function (pointed to by the `SYSENTER_EIP_MSR` register).
4. The `sysenter_entry( )` assembly language function performs the following steps:
   a. Sets up the kernel stack pointer:
   b.
   ```
   movl -508(%esp), %esp
   ```

      Initially, the `esp` register points to the first location after the local TSS, which is 512bytes long. Therefore, the instruction loads in the `esp` register the contents of the field at offset 4 in the local TSS, that is, the contents of the `esp0` field. As already explained, the `esp0` field always stores the kernel stack pointer of the current process.

   c. Enables local interrupts:
   d.
   ```
   sti
   ```

   e. Saves in the Kernel Mode stack the Segment Selector of the user data segment, the current user stack pointer, the `eflags` register, the Segment Selector of the user code segment, and the address of the instruction to be executed when exiting from the system call:
   ```
   f.        pushl $(__USER_DS)
   g.        pushl %ebp
   h.        pushfl
   i.        pushl $(__USER_CS)
   j.        pushl $SYSENTER_RETURN
   ```

      Observe that these instructions emulate some operations performed by the `int` assembly language instruction (steps 5c and 7 in the description of `int` in the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4).

   k. Restores in `ebp` the original value of the register passed by the wrapper routine:

   ```
   movl (%ebp), %ebp
   ```

This instruction does the job, because _ _kernel_vsyscall( ) saved on the User Mode stack the original value of ebp and then loaded in ebp the current value of the user stack pointer.

l. Invokes the system call handler by executing a sequence of instructions identical to that starting at the system_call label described in the earlier section "Issuing a System Call via the int $0x80 Instruction."

10.3.2.4. Exiting from the system call

When the system call service routine terminates, the sysenter_entry( ) function executes essentially the same operations as the system_call( ) function (see previous section). First, it gets the return code of the system call service routine from eax and stores it in the kernel stack location where the User Mode value of the eax register is saved. Then, the function disables the local interrupts and checks the flags in the thread_info structure of current.

If any of the flags is set, then there is some work to be done before returning to User Mode. In order to avoid code duplication, this case is handled exactly as in the system_call( ) function, thus the function jumps to the resume_userspace or work_pending labels (see flow diagram in Figure 4-6 in Chapter 4). Eventually, the iret assembly language instruction fetches from the Kernel Mode stack the five arguments saved in step 4c by the sysenter_entry( ) function, and thus switches the CPU back to User Mode and starts executing the code at the SYSENTER_RETURN label (see below).

If the sysenter_entry( ) function determines that the flags are cleared, it performs a quick return to User Mode:

```
movl 40(%esp), %edx
movl 52(%esp), %ecx
xorl %ebp, %ebp
sti
sysexit
```

The edx and ecx registers are loaded with a couple of the stack values saved by sysenter_entry( ) in step 4c in the previos section: edx gets the address of the SYSENTER_RETURN label, while ecx gets the current user data stack pointer.

10.3.2.5. The sysexit instruction

The sysexit assembly language instruction is the companion of sysenter: it allows a fast switch from Kernel Mode to User Mode. When the instruction is executed, the CPU control unit performs the following steps:

1. Adds 16 to the value in the SYSENTER_CS_MSR register, and loads the result in the cs register.
2. Copies the content of the edx register into the eip register.
3. Adds 24 to the value in the SYSENTER_CS_MSR register, and loads the result in the ss register.
4. Copies the content of the ecx register into the esp register.

Because the SYSENTER_CS_MSR register is loaded with the Segment Selector of the kernel code, the cs register is loaded with the Segment Selector of the user code, while the ss register is loaded with the Segment Selector of the user data segment (see the section "The Linux GDT" in Chapter 2).

As a result, the CPU switches from Kernel Mode to User Mode and starts executing the instruction whose address is stored in the edx register.

10.3.2.6. The SYSENTER_RETURN code

The code at the SYSENTER_RETURN label is stored in the vsyscall page, and it is executed when a system call entered via sysenter is being terminated, either by the iret instruction or the sysexit instruction.

The code simply restores the original contents of the ebp, edx, and ecx registers saved in the User Mode stack, and returns the control to the wrapper routine in the standard library:

```
SYSENTER_RETURN:
  popl %ebp
  popl %edx
  popl %ecx
  ret
```

10.4. Parameter Passing

Like ordinary functions, system calls often require some input/output parameters, which may consist of actual values (i.e., numbers), addresses of variables in the address space of the User Mode process, or even addresses of data structures including pointers to User Mode functions (see the section "System Calls Related to Signal Handling" in Chapter 11).

Because the system_call( ) and the sysenter_entry( ) functions are the common entry points for all system calls in Linux, each of them has at least one parameter: the system call number passed in the eax register. For instance, if an application program invokes the fork( ) wrapper routine, the eax register is set to 2 (i.e., __NR_fork) before executing the int $0x80 or sysenter assembly language instruction. Because the register is set by the wrapper routines included in the *libc* library, programmers do not usually care about the system call number.

The fork( ) system call does not require other parameters. However, many system calls do require additional parameters, which must be explicitly passed by the

application program. For instance, the `mmap( )` system call may require up to six additional parameters (besides the system call number).

The parameters of ordinary C functions are usually passed by writing their values in the active program stack (either the User Mode stack or the Kernel Mode stack). Because system calls are a special kind of function that cross over from user to kernel land, neither the User Mode or the Kernel Mode stacks can be used. Rather, system call parameters are written in the CPU registers before issuing the system call. The kernel then copies the parameters stored in the CPU registers onto the Kernel Mode stack before invoking the system call service routine, because the latter is an ordinary C function.

Why doesn't the kernel copy parameters directly from the User Mode stack to the Kernel Mode stack? First of all, working with two stacks at the same time is complex; second, the use of registers makes the structure of the system call handler similar to that of other exception handlers.

However, to pass parameters in registers, two conditions must be satisfied:

- The length of each parameter cannot exceed the length of a register (32 bits).[*]

  [*] We refer, as usual, to the 32-bit architecture of the 80 x 86 processors. The discussion in this section does not apply to 64-bit architectures.

- The number of parameters must not exceed six, besides the system call number passed in `eax`, because 80 x 86 processors have a very limited number of registers.

The first condition is always true because, according to the POSIX standard, large parameters that cannot be stored in a 32-bit register must be passed by reference. A typical example is the `settimeofday( )` system call, which must read a 64-bit structure.

However, system calls that require more than six parameters exist. In such cases, a single register is used to point to a memory area in the process address space that contains the parameter values. Of course, programmers do not have to care about this workaround. As with every C function call, parameters are automatically saved on the stack when the wrapper routine is invoked. This routine will find the appropriate way to pass the parameters to the kernel.

The registers used to store the system call number and its parameters are, in increasing order, `eax` (for the system call number), `ebx`, `ecx`, `edx`, `esi`, `edi`, and `ebp`. As seen before, `system_call( )` and `sysenter_entry( )` save the values of these registers on the Kernel Mode stack by using the `SAVE_ALL` macro. Therefore, when the system call service routine goes to the stack, it finds the return address to `system_call( )` or to `sysenter_entry( )`, followed by the parameter stored in `ebx` (the first parameter of the system call), the parameter stored in `ecx`, and so on (see the section "Saving the registers for the interrupt handler" in Chapter 4). This stack configuration is exactly the same as in an ordinary function call, and therefore the service routine can easily refer to its parameters by using the usual C-language constructs.

Let's look at an example. The `sys_write( )` service routine, which handles the `write( )` system call, is declared as:

```
int sys_write (unsigned int fd, const char * buf, unsigned int
count)
```

The C compiler produces an assembly language function that expects to find the `fd`, `buf`, and `count` parameters on top of the stack, right below the return address, in the locations used to save the contents of the `ebx`, `ecx`, and `edx` registers, respectively.

In a few cases, even if the system call doesn't use any parameters, the corresponding service routine needs to know the contents of the CPU registers right before the system call was issued. For example, the `do_fork( )` function that implements `fork( )` needs to know the value of the registers in order to duplicate them in the child process `thread` field (see the section "The thread field" in Chapter 3). In these cases, a single parameter of type `pt_regs` allows the service routine to access the values saved in the Kernel Mode stack by the `SAVE_ALL` macro (see the section "The do_IRQ( ) function" in Chapter 4):

```
int sys_fork (struct pt_regs regs)
```

The return value of a service routine must be written into the `eax` register. This is automatically done by the C compiler when a `return n;` instruction is executed.

10.4.1. Verifying the Parameters

All system call parameters must be carefully checked before the kernel attempts to satisfy a user request. The type of check depends both on the system call and on the specific parameter. Let's go back to the `write( )` system call introduced before: the `fd` parameter should be a file descriptor that identifies a specific file, so `sys_write( )` must check whether `fd` really is a file descriptor of a file previously opened and whether the process is allowed to write into it (see the section "File-Handling System Calls" in Chapter 1). If any of these conditions are not true, the handler must return a negative valuein this case, the error code `-EBADF`.

One type of checking, however, is common to all system calls. Whenever a parameter specifies an address, the kernel must check whether it is inside the process address space. There are two possible ways to perform this check:

- Verify that the linear address belongs to the process address space and, if so, that the memory region including it has the proper access rights.
- Verify just that the linear address is lower than `PAGE_OFFSET` (i.e., that it doesn't fall within the range of interval addresses reserved to the kernel).

Early Linux kernels performed the first type of checking. But it is quite time consuming because it must be executed for each address parameter included in a system call; furthermore, it is usually pointless because faulty programs are not very common.

Therefore, starting with Version 2.2, Linux employs the second type of checking. This is much more efficient because it does not require any scan of the process memory region descriptors. Obviously, this is a very coarse check: verifying that the linear address is smaller than PAGE_OFFSET is a necessary but not sufficient condition for its validity. But there's no risk in confining the kernel to this limited kind of check because other errors will be caught later.

The approach followed is thus to defer the real checking until the last possible momentthat is, until the Paging Unit translates the linear address into a physical one. We will discuss in the section "Dynamic Address Checking: The Fix-up Code," later in this chapter, how the Page Fault exception handler succeeds in detecting those bad addresses issued in Kernel Mode that were passed as parameters by User Mode processes.

One might wonder at this point why the coarse check is performed at all. This type of checking is actually crucial to preserve both process address spaces and the kernel address space from illegal accesses. We saw in Chapter 2 that the RAM is mapped starting from PAGE_OFFSET. This means that kernel routines are able to address all pages present in memory. Thus, if the coarse check were not performed, a User Mode process might pass an address belonging to the kernel address space as a parameter and then be able to read or write every page present in memory without causing a Page Fault exception.

The check on addresses passed to system calls is performed by the access_ok( ) macro, which acts on two parameters: addr and size. The macro checks the address interval delimited by addr and addr + size - 1. It is essentially equivalent to the following C function:

```
int access_ok(const void * addr, unsigned long size)
{
    unsigned long a = (unsigned long) addr;
    if (a + size < a ||
        a + size > current_thread_info( )->addr_limit.seg)
        return 0;
    return 1;
}
```

The function first verifies whether addr + size, the highest address to be checked, is larger than 2^{32}-1; because unsigned long integers and pointers are represented by the GNU C compiler (gcc) as 32-bit numbers, this is equivalent to checking for an overflow condition. The function also checks whether addr + size exceeds the value stored in the addr_limit.seg field of the thread_info structure of current. This field usually has the value PAGE_OFFSET for normal processes and the value 0xffffffff for kernel threads . The value of the addr_limit.seg field can be dynamically changed by the get_fs and set_fs macros; this allows the kernel to bypass the security checks made by access_ok( ), so that it can invoke system call service routines, directly passing to them addresses in the kernel data segment.

The verify_area( ) function performs the same check as the access_ok( ) macro; although this function is considered obsolete, it is still widely used in the source code.

10.4.2. Accessing the Process Address Space

System call service routines often need to read or write data contained in the process's address space. Linux includes a set of macros that make this access easier. We'll describe two of them, called `get_user( )` and `put_user( )`. The first can be used to read 1, 2, or 4 consecutive bytes from an address, while the second can be used to write data of those sizes into an address.

Each function accepts two arguments, a value `x` to transfer and a variable `ptr`. The second variable also determines how many bytes to transfer. Thus, in `get_user(x,ptr)`, the size of the variable pointed to by `ptr` causes the function to expand into a `_ _get_user_1( )`, `_ _get_user_2( )`, or `_ _get_user_4( )` assembly language function. Let's consider one of them, `_ _get_user_2( )`:

```
     _ _get_user_2:
        addl $1, %eax
        jc bad_get_user
        movl $0xffffe000, %edx /* or 0xfffff000 for 4-KB stacks */
        andl %esp, %edx
        cmpl 24(%edx), %eax
        jae bad_get_user
    2:  movzwl
 -1(%eax), %edx
        xorl %eax, %eax
        ret
    bad_get_user:
        xorl %edx, %edx
        movl $-EFAULT, %eax
        ret
```

The `eax` register contains the address `ptr` of the first byte to be read. The first six instructions essentially perform the same checks as the `access_ok( )` macro: they ensure that the 2 bytes to be read have addresses less than 4 GB as well as less than the `addr_limit.seg` field of the `current` process. (This field is stored at offset 24 in the `thread_info` structure of `current`, which appears in the first operand of the `cmpl` instruction.)

If the addresses are valid, the function executes the `movzwl` instruction to store the data to be read in the two least significant bytes of `edx` register while setting the high-order bytes of `edx` to 0; then it sets a 0 return code in `eax` and terminates. If the addresses are not valid, the function clears `edx`, sets the `-EFAULT` value into `eax`, and terminates.

The `put_user(x,ptr)` macro is similar to the one discussed before, except it writes the value `x` into the process address space starting from address `ptr`. Depending on the size of `x`, it invokes either the `_ _put_user_asm( )` macro (size of 1, 2, or 4 bytes) or the `_ _put_user_u64( )` macro (size of 8 bytes). Both macros return the value 0 in the `eax` register if they succeed in writing the value, and `-EFAULT` otherwise.

Several other functions and macros are available to access the process address space in Kernel Mode; they are listed in Table 10-1. Notice that many of them also have a variant prefixed by two underscores (_ _). The ones without initial underscores take extra time to check the validity of the linear address interval requested, while the ones with the underscores bypass that check. Whenever the kernel must repeatedly access the same memory area in the process address space, it is more efficient to check the address once at the start and then access the process area without making any further checks.

Table 10-1. Functions and macros that access the process address space

Function	Action
`get_user _ _get_user`	Reads an integer value from user space (1, 2, or 4 bytes)
`put_user _ _put_user`	Writes an integer value to user space (1, 2, or 4 bytes)
`copy_from_user _ _copy_from_user`	Copies a block of arbitrary size from user space
`copy_to_user _ _copy_to_user`	Copies a block of arbitrary size to user space
`strncpy_from_user _ _strncpy_from_user`	Copies a null-terminated string from user space
`strlen_user strnlen_user`	Returns the length of a null-terminated string in user space
`clear_user _ _clear_user`	Fills a memory area in user space with zeros

10.4.3. Dynamic Address Checking: The Fix-up Code

As seen previously, `access_ok( )` makes a coarse check on the validity of linear addresses passed as parameters of a system call. This check only ensures that the User Mode process is not attempting to fiddle with the kernel address space; however, the linear addresses passed as parameters still might not belong to the process address space. In this case, a Page Fault exception will occur when the kernel tries to use any of such bad addresses.

Before describing how the kernel detects this type of error, let's specify the three cases in which Page Fault exceptions may occur in Kernel Mode. These cases must be distinguished by the Page Fault handler, because the actions to be taken are quite different.

1. The kernel attempts to address a page belonging to the process address space, but either the corresponding page frame does not exist or the kernel tries to write a read-only page. In these cases, the handler must allocate and initialize a new page frame (see the sections "Demand Paging" and "Copy On Write" in Chapter 9).
2. The kernel addresses a page belonging to its address space, but the corresponding Page Table entry has not yet been initialized (see the section

"Handling Noncontiguous Memory Area Accesses" in Chapter 9). In this case, the kernel must properly set up some entries in the Page Tables of the current process.

3. Some kernel functions include a programming bug that causes the exception to be raised when that program is executed; alternatively, the exception might be caused by a transient hardware error. When this occurs, the handler must perform a kernel oops (see the section "Handling a Faulty Address Inside the Address Space" in Chapter 9).
4. The case introduced in this chapter: a system call service routine attempts to read or write into a memory area whose address has been passed as a system call parameter, but that address does not belong to the process address space.

The Page Fault handler can easily recognize the first case by determining whether the faulty linear address is included in one of the memory regions owned by the process. It is also able to detect the second case by checking whether the corresponding master kernel Page Table entry includes a proper non-null entry that maps the address. Let's now explain how the handler distinguishes the remaining two cases.

10.4.4. The Exception Tables

The key to determining the source of a Page Fault lies in the narrow range of calls that the kernel uses to access the process address space. Only the small group of functions and macros described in the previous section are used to access this address space; thus, if the exception is caused by an invalid parameter, the instruction that caused it *must* be included in one of the functions or else be generated by expanding one of the macros. The number of the instructions that address user space is fairly small.

Therefore, it does not take much effort to put the address of each kernel instruction that accesses the process address space into a structure called the *exception table*. If we succeed in doing this, the rest is easy. When a Page Fault exception occurs in Kernel Mode, the `do_ page_fault( )` handler examines the exception table: if it includes the address of the instruction that triggered the exception, the error is caused by a bad system call parameter; otherwise, it is caused by a more serious bug.

Linux defines several exception tables . The main exception table is automatically generated by the C compiler when building the kernel program image. It is stored in the `_ _ex_table` section of the kernel code segment, and its starting and ending addresses are identified by two symbols produced by the C compiler: `_ _start_ _ _ex_table` and `_ _stop_ _ _ex_table`.

Moreover, each dynamically loaded module of the kernel (see Appendix B) includes its own local exception table. This table is automatically generated by the C compiler when building the module image, and it is loaded into memory when the module is inserted in the running kernel.

Each entry of an exception table is an `exception_table_entry` structure that has two fields:

`insn`

The linear address of an instruction that accesses the process address space

`fixup`

The address of the assembly language code to be invoked when a Page Fault exception triggered by the instruction located at `insn` occurs

The fixup code consists of a few assembly language instructions that solve the problem triggered by the exception. As we will see later in this section, the fix usually consists of inserting a sequence of instructions that forces the service routine to return an error code to the User Mode process. These instructions, which are usually defined in the same macro or function that accesses the process address space, are placed by the C compiler into a separate section of the kernel code segment called `.fixup`.

The `search_exception_tables( )` function is used to search for a specified address in all exception tables: if the address is included in a table, the function returns a pointer to the corresponding `exception_table_entry` structure; otherwise, it returns `NULL`. Thus the Page Fault handler `do_page_fault( )` executes the following statements:

```
if ((fixup = search_exception_tables(regs->eip))) {
    regs->eip = fixup->fixup;
    return 1;
}
```

The `regs->eip` field contains the value of the `eip` register saved on the Kernel Mode stack when the exception occurred. If the value in the register (the instruction pointer) is in an exception table, `do_page_fault( )` replaces the saved value with the address found in the entry returned by `search_exception_tables( )`. Then the Page Fault handler terminates and the interrupted program resumes with execution of the fixup code .

10.4.5. Generating the Exception Tables and the Fixup Code

The GNU Assembler `.section` directive allows programmers to specify which section of the executable file contains the code that follows. As we will see in Chapter 20, an executable file includes a code segment, which in turn may be subdivided into sections. Thus, the following assembly language instructions add an entry into an exception table; the `"a"` attribute specifies that the section must be loaded into memory together with the rest of the kernel image:

```
.section _ _ex_table, "a"
    .long faulty_instruction_address, fixup_code_address
```

```
.previous
```

The `.previous` directive forces the assembler to insert the code that follows into the section that was active when the last `.section` directive was encountered.

Let's consider again the `_ _get_user_1( )`, `_ _get_user_2( )`, and `_ _get_user_4( )` functions mentioned before. The instructions that access the process address space are those labeled as `1`, `2`, and `3`:

```
_ _get_user_1:
    [...]
1:  movzbl (%eax), %edx
    [...]
_ _get_user_2:
    [...]
2:  movzwl -1(%eax), %edx
    [...]
_ _get_user_4:
    [...]
3:  movl -3(%eax), %edx
    [...]
bad_get_user:
    xorl %edx, %edx
    movl $-EFAULT, %eax
    ret
.section _ _ex_table,"a"
    .long 1b, bad_get_user
    .long 2b, bad_get_user
    .long 3b, bad_get_user
.previous
```

Each exception table entry consists of two labels. The first one is a numeric label with a `b` suffix to indicate that the label is "backward;" in other words, it appears in a previous line of the program. The fixup code is common to the three functions and is labeled as `bad_get_user`. If a Page Fault exception is generated by the instructions at label `1`, `2`, or `3`, the fixup code is executed. It simply returns an `-EFAULT` error code to the process that issued the system call.

Other kernel functions that act in the User Mode address space use the fixup code technique. Consider, for instance, the `strlen_user(string)` macro. This macro returns either the length of a null-terminated string passed as a parameter in a system call or the value 0 on error. The macro essentially yields the following assembly language instructions:

```
    movl $0, %eax
    movl $0x7fffffff, %ecx
    movl %ecx, %ebx
    movl string, %edi
0:  repne; scasb

    subl %ecx, %ebx
```

```
    movl %ebx, %eax
1:
.section .fixup,"ax"
2:  xorl %eax, %eax
    jmp 1b
.previous
.section _ _ex_table,"a"
    .long 0b, 2b
.previous
```

The ecx and ebx registers are initialized with the 0x7fffffff value, which represents the maximum allowed length for the string in the User Mode address space. The repne;scasb assembly language instructions iteratively scan the string pointed to by the edi register, looking for the value 0 (the end of string \0 character) in eax. Because scasb decreases the ecx register at each iteration, the eax register ultimately stores the total number of bytes scanned in the string (that is, the length of the string).

The fixup code of the macro is inserted into the .fixup section. The "ax" attributes specify that the section must be loaded into memory and that it contains executable code. If a Page Fault exception is generated by the instructions at label 0, the fixup code is executed; it simply loads the value 0 in eaxthus forcing the macro to return a 0 error code instead of the string lengthand then jumps to the 1 label, which corresponds to the instruction following the macro.

The second .section directive adds an entry containing the address of the repne; scasb instruction and the address of the corresponding fixup code in the _ _ex_table section.

10.5. Kernel Wrapper Routines

Although system calls are used mainly by User Mode processes, they can also be invoked by kernel threads , which cannot use library functions. To simplify the declarations of the corresponding wrapper routines , Linux defines a set of seven macros called _syscall0 through _syscall6.

In the name of each macro, the numbers 0 through 6 correspond to the number of parameters used by the system call (excluding the system call number). The macros are used to declare wrapper routines that are not already included in the *libc* standard library (for instance, because the Linux system call is not yet supported by the library); however, they cannot be used to define wrapper routines for system calls that have more than six parameters (excluding the system call number) or for system calls that yield nonstandard return values.

Each macro requires exactly 2 + 2 x *n* parameters, with *n* being the number of parameters of the system call. The first two parameters specify the return type and the name of the system call; each additional pair of parameters specifies the type and the name of the corresponding system call parameter. Thus, for instance, the wrapper routine of the fork( ) system call may be generated by:

```
_syscall0(int,fork)
```

while the wrapper routine of the write( ) system call may be generated by:

```
_syscall3(int,write,int,fd,const char *,buf,unsigned int,count)
```

In the latter case, the macro yields the following code:

```
int write(int fd,const char * buf,unsigned int count)
{
    long _ _res;
    asm("int $0x80"
        : "=a" (_ _res)
        : "0" (_ _NR_write), "b" ((long)fd),
          "c" ((long)buf), "d" ((long)count));
    if ((unsigned long)_ _res >= (unsigned long)-129) {
        errno = -_ _res;
        _ _res = -1;
    }
    return (int) _ _res;
}
```

The _ _NR_write macro is derived from the second parameter of _syscall3; it expands into the system call number of write( ). When compiling the preceding function, the following assembly language code is produced:

```
write:
     pushl %ebx                ; push ebx into stack
     movl 8(%esp), %ebx        ; put first parameter in ebx
     movl 12(%esp), %ecx       ; put second parameter in ecx
     movl 16(%esp), %edx       ; put third parameter in edx
     movl $4, %eax             ; put _ _NR_write in eax
     int
 $0x80                ; invoke system call
     cmpl $-125, %eax          ; check return code
     jbe .L1                   ; if no error, jump
     negl %eax                 ; complement the value of eax
     movl %eax, errno          ; put result in errno
     movl $-1, %eax            ; set eax to -1
.L1: popl %ebx                 ; pop ebx from stack
     ret                       ; return to calling program
```

Notice how the parameters of the write( ) function are loaded into the CPU registers before the int $0x80 instruction is executed. The value returned in eax must be interpreted as an error code if it lies between -1 and -129 (the kernel assumes that the largest error code defined in *include/generic/errno.h* is 129). If this is the case, the wrapper routine stores the value of -eax in errno and returns the value -1; otherwise, it returns the value of eax.