System Calls

Introduction

- Operating systems offer processes running in User Mode a set of interfaces to interact with hardware devices such as the CPU, disks, printers, and so on.
- Unix systems implement most interfaces between User Mode processes and hardware devices by means of system calls issued to the kernel.

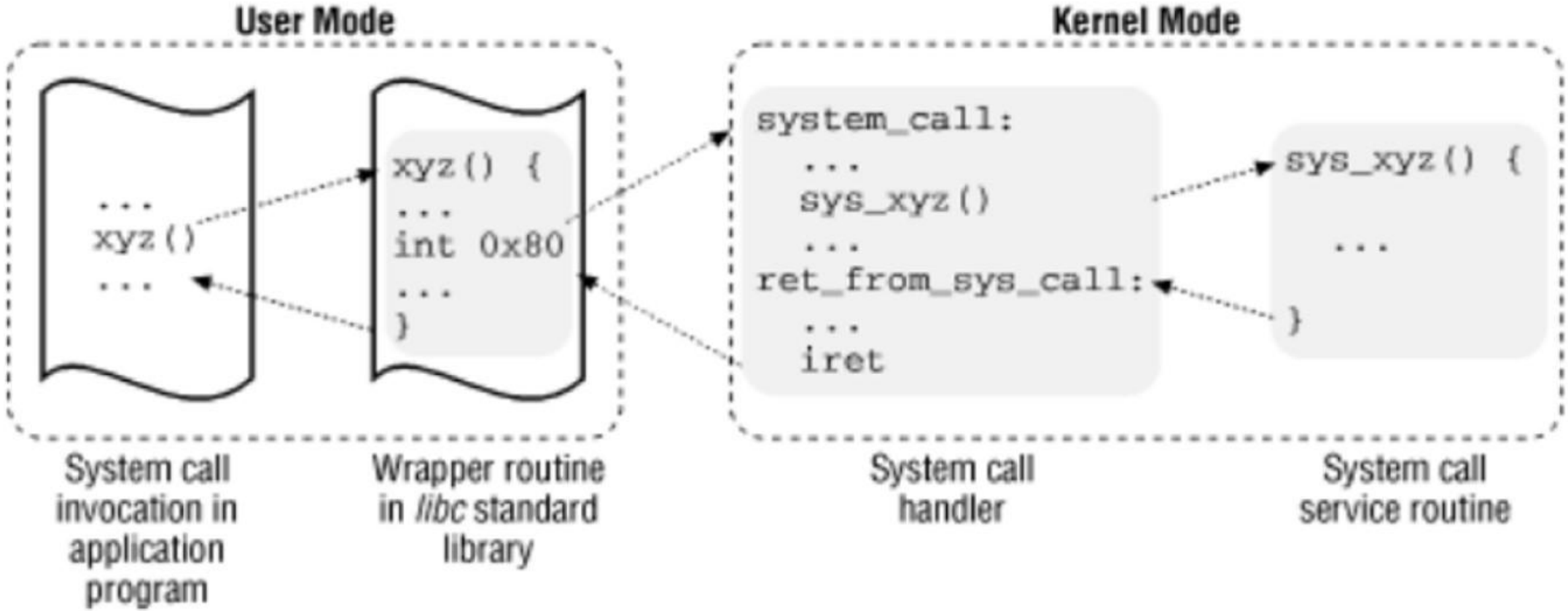
POSIX APIs and System call

- Difference between API and System call.
- Function definition specifying how to obtain a given service. Explicit request to the kernel made via a software interrupt.
- An API does not necessarily correspond to a specific system call, the API could offer its services directly in User Mode.
- A single API function could make several system calls. Several API functions could make the same system call but wrap extra functionality around it. Eg – malloc(), calloc(), and free().
- POSIX APIs are implemented in the libc library: the code in that library keeps track of the allocation and deallocation requests and uses the brk( ) system call in order to enlarge or shrink the process heap.
- The POSIX standard refers to APIs and not to system calls.
- From the programmer's point of view, the function name, the parameter types, and the meaning of the return code matters most rather than distinction.

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.
- In Linux the system calls must be invoked by executing the int $0x80 Assembly instruction, which raises the programmed exception having vector 128.
- Since the kernel implements many different system calls, the process must pass a parameter called the system call number to identify the required system call; the eax register is used for that purpose.
- All system calls return an integer value.
- In the kernel, positive or null values denote a successful termination of the system call, while negative values denote an error condition.

- The system call handler 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
  - Handles the system call by invoking a corresponding C function called the system call service routine.
  - Exits from the handler by means of the ret_from_sys_call( ) function.



- In order to associate each system call number with its corresponding service routine, the kernel makes use of a system call dispatch table.
- This table is stored in the sys_call_table array and has NR_syscalls entries (usually 256): 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.

Initializing System Calls

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

- The call loads the following values into the gate descriptor fields:
  - Segment Selector - The __KERNEL_CS Segment Selector of the kernel code segment.
  - Offset - Pointer to the system_call( ) exception handler.
  - Type - Set to 15. Indicates that the exception is a Trap and that the corresponding handler does not disable maskable interrupts.
  - DPL - Set to 3; this allows processes in User Mode to invoke the exception handler.

The system_call() Function

- The system_call( ) function implements the system call handler. It starts by saving the system call number and all the CPU registers that may be used by the exception handler on the stack, except for eflags, cs, eip, ss, and esp, which have already been saved automatically by the control unit.

```
system_call:
pushl %eax
SAVE_ALL
movl %esp, %ebx
andl $0xffffe000,
%ebx
```

Parameter passing

- Like ordinary functions, system calls often require some input/output parameters, which may consist of actual values (i.e., numbers) or addresses of functions and variables in the address space of the User Mode process.
- Since the system_call( ) function is the unique entry point for all system calls in Linux, each of them has at least one parameter: the system call number passed in the eax register.
- The fork( ) system call does not require other parameters, whereas the mmap( ) system call may require up to six parameters.
- System call parameters are usually passed to the system call handler in the CPU registers, then copied onto the Kernel Mode stack, since system call service routines are ordinary C functions.

- In order to pass parameters in registers, two conditions must be satisfied:
- The length of each parameter cannot exceed the length of a register, that is 32 bits.
- The number of parameters must not exceed six.

Verifying the parameters

- 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).

Wrapper routines

- Although system calls are mainly used by User Mode processes, they can also be invoked by kernel threads, which cannot make use of library functions.
- In order to simplify the declarations of the corresponding wrapper routines, Linux defines a set of six macros called _syscall0 through _syscall5.

```
_syscall0(int,fork)

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

- Each macro requires exactly 2+2xn 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.