FRANKFURT UNIVERSITY OF APPLIED SCIENCES

OPERATING SYSTEMS

System Calls

Prof. Dr. Oliver Hahm

2024-12-05

AGENDA

- Privilege Levels
- System Calls
- System Call: read()

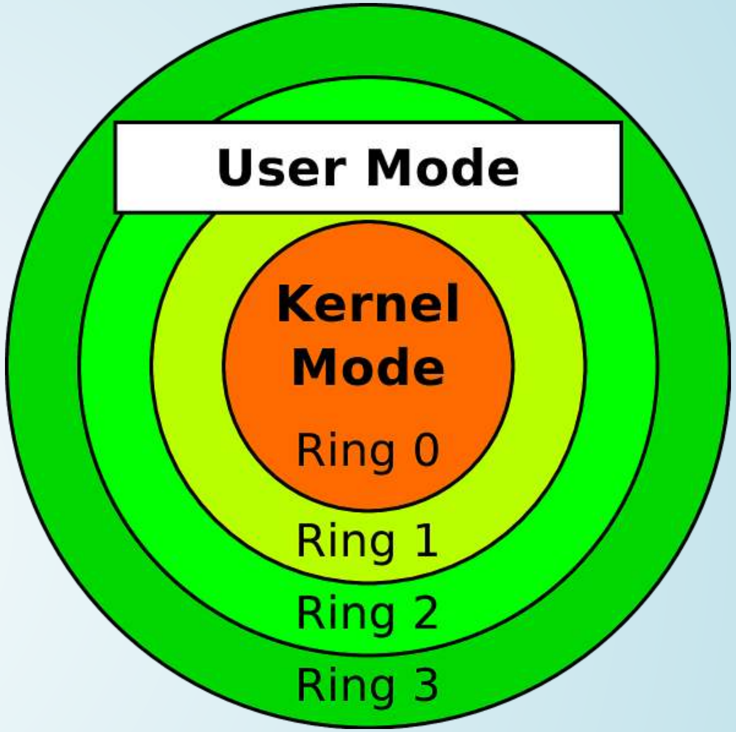
PRIVILEGE LEVELS

RESTRICT PROCESSES

How can we restrict processes?
For example, how can we prevent user mode processes
to access memory directly?

USER MODE AND KERNEL MODE

- x86-compatible CPUs implement four privilege levels
  - **Objective**: Improve stability and security
  - Each process is assigned to a ring permanently (stored in register CPL (Current Privilege Level))
- Ring 0 (= **kernel mode**) runs the kernel
  - processes have full access to the hardware
  - The kernel can also address physical memory (⟶ Real Mode)
- Ring 3 (= **user mode**) run the applications
  - processes can only access virtual memory (⟶ Protected Mode)



Modern operating systems use only two privilege levels (rings)

Reason: Some hardware architectures (e.g., Alpha, PowerPC, MIPS) implement only two levels
Consequence: Intel's most recent x86-s architecture removes ring 1 and 2

SYSTEM CALLS

How can a process from user space access the hardware?

SYSTEM CALLS

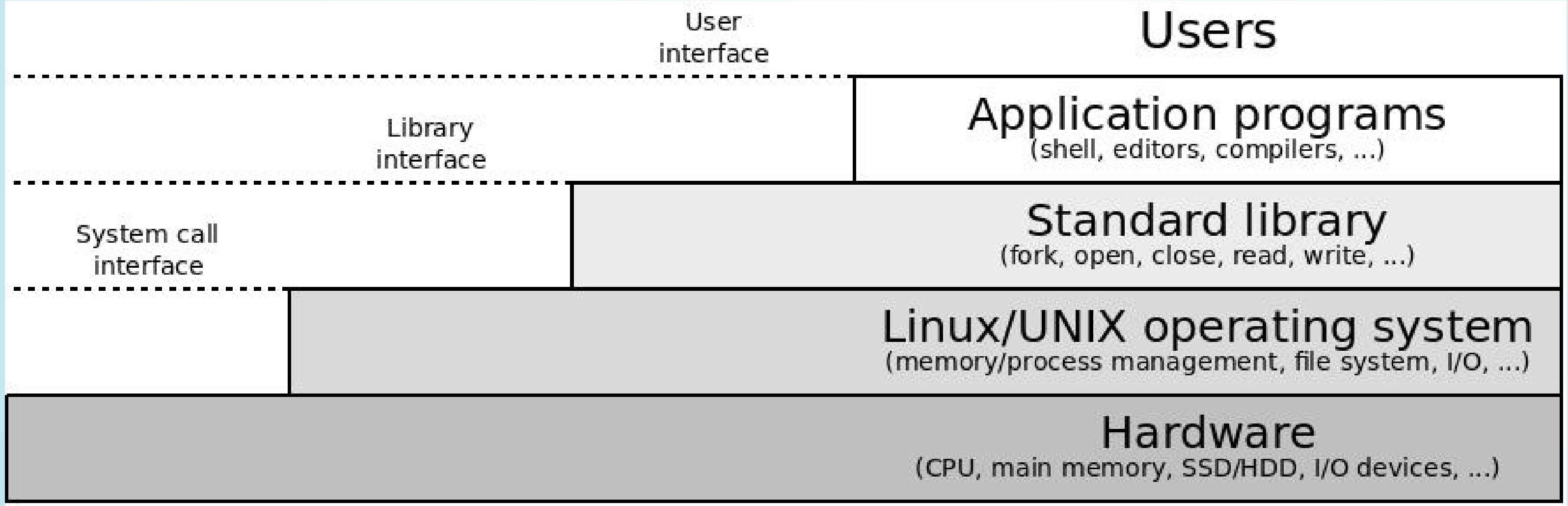
- If a user-mode process must carry out a higher privileged task (e.g., access hardware), it can tell this the kernel via a system call
  - A system call is a function call in the operating system that triggers a switch from user mode to kernel mode

Context Switch

- A process passes the control over the CPU to the kernel and is suspended until the request is completely processed
- After the system call, the kernel returns the control over the CPU to the user-mode process
- The process continues its execution at the point, where the context switch was previously requested

SYSTEM CALL INTERFACE

- System calls are the interface, which provides the operating system to the user mode processes
  - System calls enable the user mode programs among others to create and manage processes and files and to access the hardware



In other words:

A system call is a request from a user mode process to the kernel in order to use a service of the kernel

EXAMPLE OF A SYSTEM CALL: `ioctl()`

- In Unix-like OS (e.g., Linux) `ioctl()` allows programs to **control** the behavior of **I/O devices**
  - `ioctl()` enables processes to communicate with and control of:
    - Character devices (Mouse, keyboard, printer, terminals, ...)
    - Block devices (SSD/HDD, CD/DVD drive, ...)
- Syntax:

```
ioctl (File descriptor, request code number, integer value or pointer to data);
```

- Typical application scenarios of `ioctl()`:
  - Adjust terminal settings (window size or mode)
  - Initialize peripheral devices like a sound card or camera
  - Controlling file locks
  - Socket operations
  - Retrieve status and link information of a network interface
  - Access sensors via the I^2C bus

SYSTEM CALLS AND LIBRARIES

- Working directly with system calls has two major drawbacks:
  - Missing abstractions (⇒ e.g., missing error handling)
  - Portability is poor
- Modern operating systems provide an interface towards the system calls in form of a C library, e.g.,: GNU C library (glibc) on (Linux), Native API `ntdll.dll` (Windows)

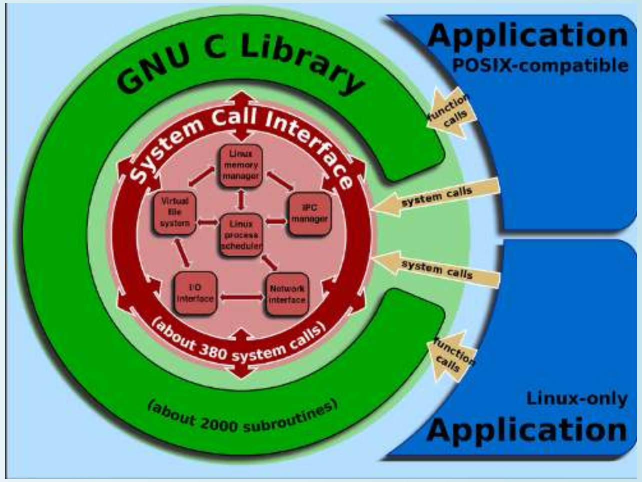


Image Source: Wikipedia
(Shmuel Csaba Otto Traian, CC-BY-SA-3.0)

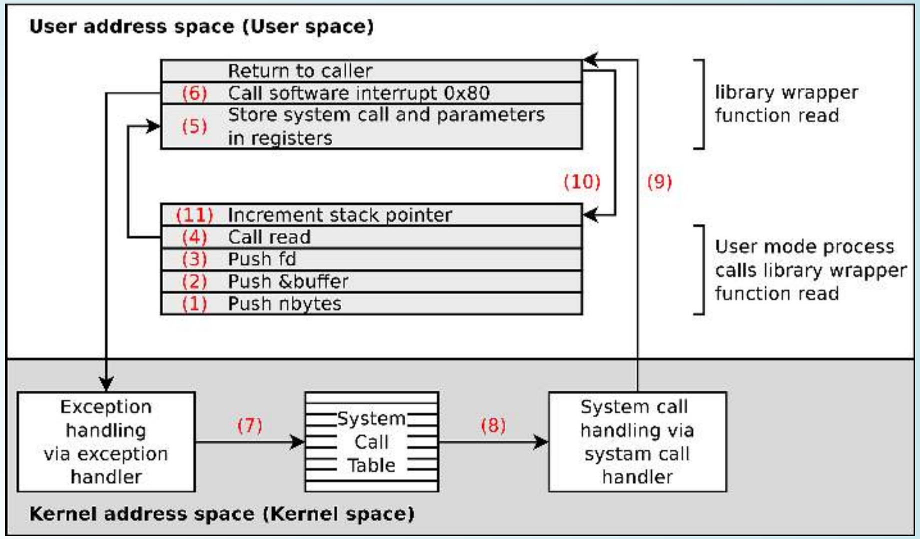
- The library is responsible for:
  - Handling the communication between user mode processes and kernel
  - Context switching between user mode and kernel mode
- Advantages which result in using a library:
  - Increased portability, because there is no or very little need for the user mode processes to communicate directly with the kernel
  - Increased security, because the user mode processes can not trigger the context switch to kernel mode for themselves

SYSTEM CALL: READ()

- If a (user mode) application wants to read data from a file, a **system call** is required
  - Before the reading call another system call, `open()` is required
  - This call returns a handle, called **file descriptor (fd)**
- The application can neither access the file system directly nor the underlying storage device
- Library system call function:
  `read(fd, buffer, nbytes);`
  - $\longrightarrow$ read `nbytes` from the file `fd` and store it inside `buffer`

STEP BY STEP – `read(fd, buffer, nbytes);`

- Step **9**: The exception handler returns control back to the library which triggered the software interrupt
- Step **10**: This function returns back to the user mode process, in the way a normal function would have done it



- Step **11**: To complete the system call, the user mode process must clean up the stack just like after every function call
- The user process can now continue to operate

Source of this example Modern Operating Systems, *Andrew S. Tanenbaum*, 3rd edition, Pearson (2009), P.84-89

EXAMPLE OF A SYSTEM CALL IN LINUX

- System calls are called like library wrapper functions
  - The mechanism is similar for all operating systems
  - In a C program, no difference is visible

```c
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
int main(void) {
    unsigned int ID1, ID2;
    // System call
    ID1 = syscall(SYS_getpid);
    printf ("Result of the system call: %d\n", ID1);
    // Wrapper function of the glibc, which calls the system call
    ID2 = getpid();
    printf ("Result of the wrapper function: %d\n", ID2);
    return(0);
}
```

```
$ gcc SysCallBeispiel.c -o SysCallBeispiel
$ ./SysCallBeispiel
Result of the system call: 3452
Result of the wrapper function: 3452
```

SELECTION OF SYSTEM CALLS

Process management | File management | Directory management | Miscellaneous

`fork`	Create a new child process
`waitpid`	Wait for the termination of a child process
`execve`	Replace a process by another one. The PID is kept
`exit`	Terminate a process

LINUX SYSTEM CALLS

- The list with the names of the system calls in the Linux kernel...
  - is located in the source code of kernel 2.6.x in the file: `arch/x86/kernel/syscall_table_32.S`
  - is located in the source code of kernel 3.x, 4.x and 5.x in these files: `arch/x86/syscalls/syscall_[64|32].tbl` or `arch/x86/entry/syscalls/syscall_[64|32].tbl`

arch/x86/syscalls/syscall_32.tbl

```
...
1       i386    exit                    sys_exit
2       i386    fork                    sys_fork
3       i386    read                    sys_read
4       i386    write                   sys_write
5       i386    open                    sys_open
6       i386    close                   sys_close
...
```

Tutorials how to implement own system calls

https://www.kernel.org/doc/html/v4.14/process/adding-syscalls.html https://brennan.io/2016/11/14/kernel-dev-ep3/
https://medium.com/@jeremyphilemon/adding-a-quick-system-call-to-the-linux-kernel-cad55b421a7b
https://medium.com/@ssreehari/implementing-a-system-call-in-linux-kernel-4-7-1-6f98250a8c38
http://tldp.org/HOWTO/Implement-Sys-Call-Linux-2.6-i386/index.html http://www.ibm.com/developerworks/library/l-system-calls/

SUMMARY

You should now be able to answer the following questions:

- How are different process privileges represented in hardware?
- How can a user mode process execute a higher privileged task?
- How is exception handling being carried out?