Operating Systems
System Calls

Prof. Dr. Oliver Hahm

Frankfurt University of Applied Sciences
Faculty 2: Computer Science and Engineering
oliver.hahm@fb2.fra-uas.de
https://teaching.dahahm.de

November 21, 2023

Agenda

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

Agenda

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

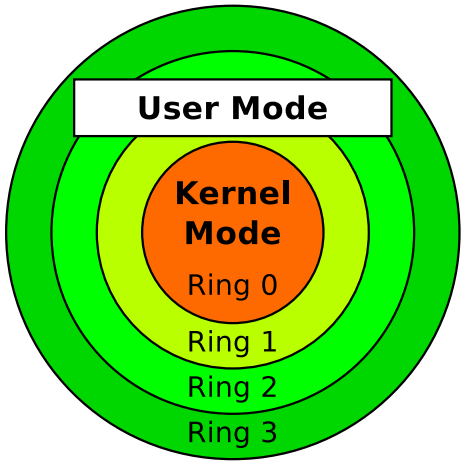
Restrict Processes

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

Agenda

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

System Calls (1/2)

How can a process from user space access the hardware?

System Calls (1/2)

How can a process from user space access the hardware?

- 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

System Calls (1/2)

How can a process from user space access the hardware?

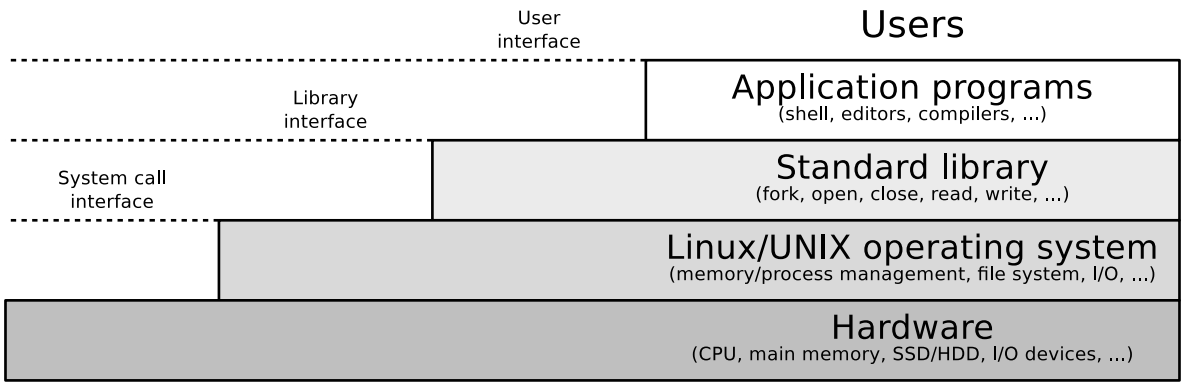
- If a user-mode process must carry out a higher privileged task (e.g., access hardware), it can tell this the kernel via a system call
  - A system call is a function call in the operating system that triggers a switch from user mode to kernel mode ($\longrightarrow$ **context switch**)

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 Calls (2/2)

- 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



Simply stated...

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 ($\Rightarrow$ e.g., missing error handling)
  - Portability is poor

$\Rightarrow$ 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)

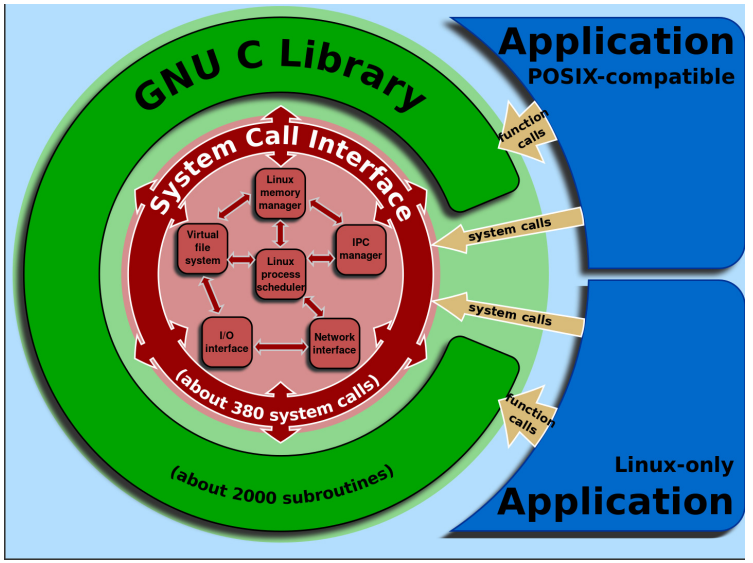


Image Source: Wikipedia
(Shmuel Csaba Otto Traian, CC-BY-SA-3.0)

- 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

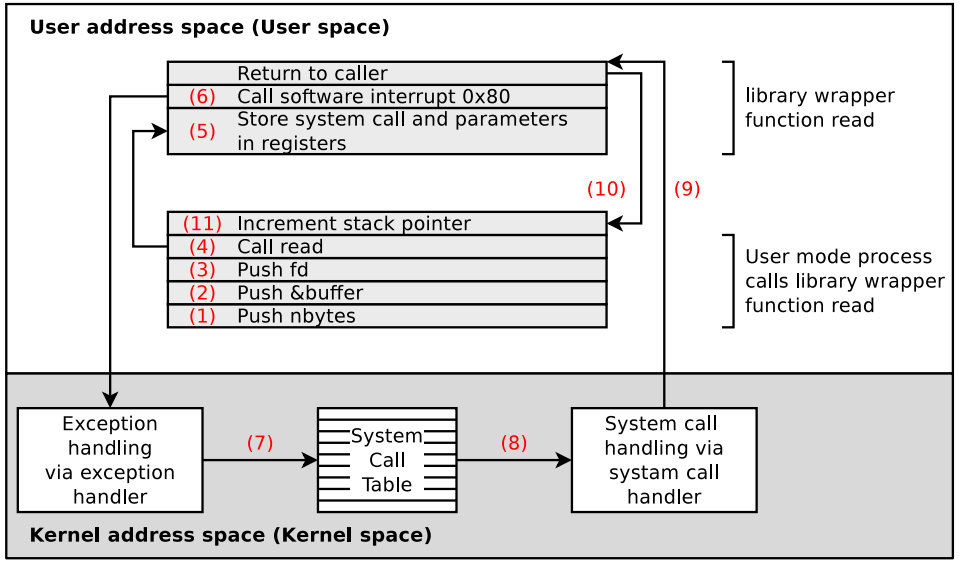
Agenda

- Privilege Levels
- System Calls
- 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

Step by Step (1/4) – `read(fd, buffer, nbytes);`

- In step 1-3 stores the user mode process the parameters on the stack
- In 4 calls the user mode process the library wrapper function for `read` ($\longrightarrow$ read `nbytes` from the file `fd` and store it inside `buffer`)



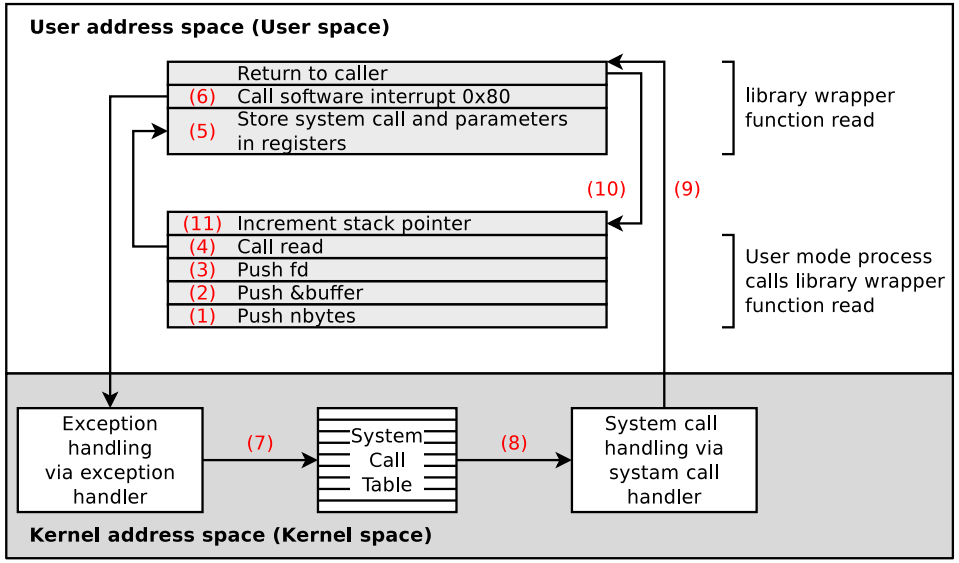
- In 5 stores the library wrapper function the system call number in the *accumulator register* EAX (32 bit) or RAX (64 bit)
  - The library wrapper function stores the parameters of the system call in the registers EBX, ECX and EDX (or for 64 bit: RBX, RCX and RDX)

Source of this example

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

Step by Step (2/4) – `read(fd, buffer, nbytes);`

- In 6, the software interrupt (exception) 0x80 is triggered to switch from **user mode** to **kernel mode**
  - An interrupt enforces the current process execution to be interrupted and calling a corresponding handler in kernel mode

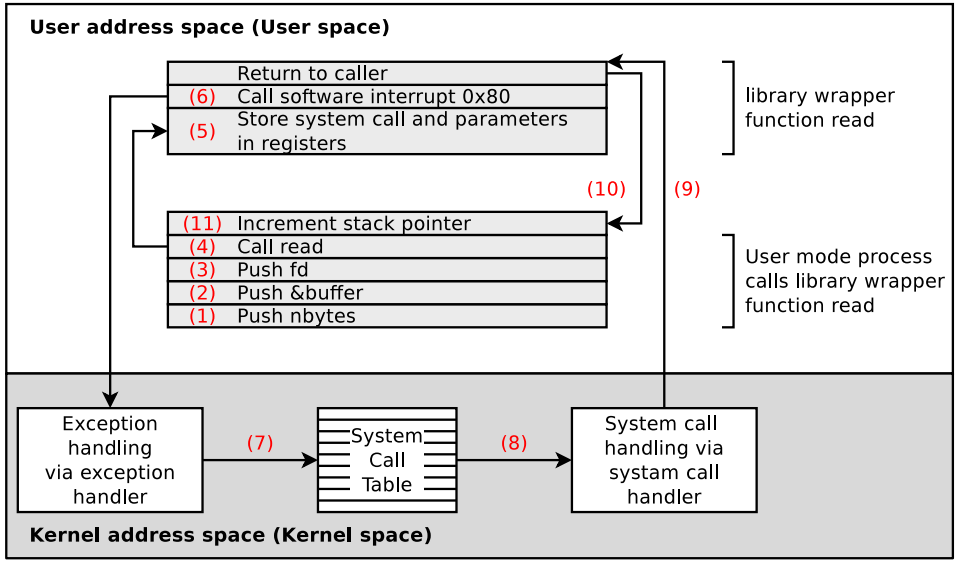


The kernel maintains the System Call Table, a list of all system calls

In this list, each system call is assigned to a unique number and an internal kernel function

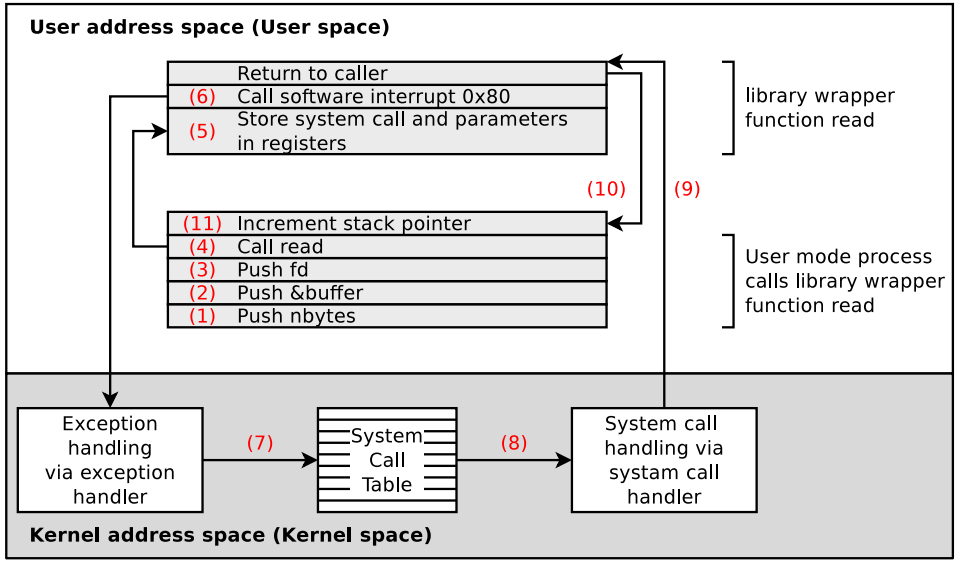
Step by Step (3/4) – `read(fd, buffer, nbytes);`

- The exception handling code of the kernel reads out the content of the EAX (resp. RAX) register
- In 7 according to the registered exception handler, the corresponding kernel function from the system call table with the arguments, which are stored in the registers EBX, ECX, and EDX (resp. RBX, RCX, and RDX)
- In 8, the actual system call is executed



Step by Step (4/4) – `read(fd, buffer, nbytes);`

- In 9, the exception handler returns control back to the library, which triggered the software interrupt
- Next, this function returns in 10 back to the user mode process, in the way a normal function would have done it



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

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	`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
File management	`open`	Open file for reading/writing
	`close`	Close an open file
	`read`	Read data from a file into the buffer
	`write`	Write data from the buffer into a file
	`lseek`	Reposition read/write file offset
	`stat`	Determine the status of a file
Directory management	`mkdir`	Create a new directory
	`rmdir`	Remove an empty directory
	`link`	Create a directory entry (link) to a file
	`unlink`	Erase a directory entry
	`mount`	Attach a file system to the file system hierarchy
	`umount`	Detatch a file system
Miscellaneous	`chdir`	Change current directory
	`chmod`	Change file permissions of a file
	`kill`	Send signal to a process
	`time`	Seconds since January 1st, 1970 („UNIX time“)

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

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?