Operating Systems 2016/17
Tutorial-Assignment 1

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Question 1.1: Basics

a. Enumerate the major tasks of an operating system.

Solution:

- *Abstraction/Standardization: An operating system should hide hardware details from applications/application programmers/users.*
- *Resource Management: Resources must be multiplexed in a "fair" way between applications/users.*
- *Security/Protection: This point is closely related to resource multiplexing. Different applications should not be able to disturb/manipulate each other. No private data of one user should be exposed to other users, unless explicitly desired.*
- *Providing an execution environment for applications: This point can be seen as the main goal of an operating system, the 3 points mentioned above are requirements to fulfill this goal.*

b. What are some of the differences between a processor running in privileged mode (also called kernel mode) and user mode? Why are the two modes needed?

Solution:
In user mode, only the non-privileged processor instructions can be executed. Executing a privileged processor instruction in user mode will lead to an exception of type privileged instruction violation. *It is up to the system programmer how to react to this exceptional event. In most cases, the corresponding exception handler will abort the activity that caused this exception.*

In kernel mode, most processors allow the execution of all instructions. On many existing systems that support only user and kernel mode you cannot prevent kernel code from accessing every system or application entity (i.e., object in memory). Since Broadwell Intel CPUs support Supervisor Mode Execution/Access Prevention (SMEP/SMAP), which prevents execution/access of user instructions/memory from kernel mode.

Some activities that control and manage the system as a whole may only be performed in kernel mode. Thus kernel mode is necessary in order to avoid that applications can tune the system to their own needs while negatively affecting other applications (e.g., by occupying system resources as long as they need them without paying for their usage).

c. What are typical examples of privileged instructions? Why are they privileged?

Solution:
Instructions that

(a) manipulate control registers for memory translation,

(b) disable or enable interrupts, or

(c) access platform devices are usually privileged.

If these instructions were not privileged, they would allow user-level software to manipulate critical system state, potentially elevate its privileges, overcome protection, ruin isolation, and compromise other system properties like stability, trustworthiness, . . .

The following is an excerpt from the IA-32 Intel® Architecture Software Developer's Manual Volume 3: System Programming Guide. The complete document can be obtained via "Developer Resources" from the Intel website.

4.9 Privileged Instructions *Some of the system instructions (called "privileged instructions") are protected from use by application programs. The privileged instructions control system functions (such as the loading of system registers). They can be executed only when the CPL is 0 (most privileged). If one of these instructions is executed when the CPL is not 0, a general-protection exception (#GP) is generated. The following system instructions are privileged instructions: • LGDT — Load GDT register. • LLDT — Load LDT register. • LTR — Load task register. • LIDT — Load IDT register. • MOV (control registers) — Load and store control registers. • LMSW — Load machine status word. • CLTS — Clear task-switched flag in register CR0. • MOV (debug registers) — Load and store debug registers. • INVD — Invalidate cache, without writeback. • WBINVD — Invalidate cache, with writeback. • INVLPG — Invalidate TLB entry. • HLT — Halt processor. • RDMSR — Read Model-Specific Registers. • WRMSR — Write Model-Specific Registers. • RDPMC — Read Performance-Monitoring Counter. • RDTSC — Read Time-Stamp Counter.*

Some of the privileged instructions are available only in the more recent families of IA-32 processors (see Section 18.12.,"New Instructions In the Pentium and Later IA-32 Processors"). The PCE and TSD flags in register CR4 (bits 4 and 2, respectively) enable the RDPMC and RDTSC instructions, respectively, to be executed at any CPL.

Question 1.2: The User/Kernel Boundary

a. The operating system does not always run (even if multiple CPUs/cores are available). What three events can lead to an invocation of the kernel?

Solution:
There are basically three ways to invoke the kernel:

- ***Interrupts*** *An interrupt is a signal that is generated outside the processor–usually by some I/O-device controller–to notify the operating system of external events (e.g., a key has been pressed, or a network packet has been received).*
- ***Exceptions*** *Exceptions are used by the CPU to inform the operating system of error conditions in the course of executing a thread. Exceptions are always a side-effect of a failed instruction such as load/store (pagefault), division by zero, or the attempted execution of a privileged instruction in user-mode. The CPU detects the error condition and traps into the operating system which is expected to handle it.*
- ***System calls*** *System calls are explicit calls from an user-application into the operating system with the intention to execute/receive some service by/from the OS (e.g., read from a file).*

b. Kernel entries can be classified in at least two dimensions:

   (a) voluntary vs. involuntary
   (b) synchronous vs. asynchronous

   Associate each of the three kernel entries with a combination of the two parameters and discuss the unassigned combination with respect to potential use in current or future systems.

   Solution:
   As seen from the perspective from the running thread, ***interrupts*** *are involuntary and asynchronous kernel entries: Neither did the interrupted thread call for the interrupt, nor did it specify when the interrupt should occur. The activity that raised the interrupt executes independently (or asynchronously) of the current thread, e.g., in a device controller.*

   Exceptions *are involuntary synchronous kernel entries. Pagefaults or divide-by-0 exceptions are typical examples of exceptions: The application caused the event in the course of its execution, a specific instruction (`load` or `div`) triggered the kernel entry synchronously with (as part of) the program's execution. However, as the application intended to load from memory or divide, the kernel entry is a mere side-effect rather than the desired action and hence regarded as involuntary.*

   System calls *are voluntary synchronous kernel entries. The kernel entry is caused by the execution of a specific instruction (such as `int`, `syscall`, or `sysenter`) and hence considered synchronous. As the whole purpose of executing these instructions is to enter the kernel, the event is voluntary.*

   The remaining combination, voluntary but asynchronous kernel entries, does not seem to have any instances: The application would have to express its desire to enter the kernel, but the time of entry would have to be left unspecified (to make it asynchronous, the actual entry must be triggered from outside the program's realm of influence).

c. Some systems try to enhance kernel protection by introducing an extra kernel stack instead of using the application stack of the currently running application whilst performing kernel operations. Does this technique really improve kernel protection? Explain!

   Solution:
   If the kernel executes on the application's stack, kernel-internal information that is not necessarily related to this application could leak into the application's reach. Additionally, using the user-mode stack severely erodes system security as other user-level threads knowing that a certain thread is executing in the kernel, can change the stack contents, leading the kernel to execute arbitrary system-level code with arbitrary input arguments! Furthermore, the application's stack may not provide sufficient space for the kernel operations. For example, the application's stack pointer might be close to the end of a page, while the next page has not been mapped in yet. On some architectures, the processor may enter shutdown state (system crash/halt) when it experiences a pagefault while trying to push an exception frame onto the stack.

d. What is an interrupt vector?

   Solution:
   An interrupt vector is either directly the entry address of an interrupt handler (e.g., on MIPS and many microcontrollers) or the index into an array of such addresses (a so-called interrupt vector table, *found for example in x86-based systems).*

e. What is an interrupt service routine (ISR)?

Solution:
An ISR is the piece of system code that handles an interrupt. ISRs are often part of a device driver, as the action that has to be taken by the OS upon an interrupt depends on the device that triggered that interrupt. Part of interrupt handling may be reading or writing device registers and device memory.

f. Some exceptions usually lead to the abortion of the executing thread/task while others can be repaired by the OS. Find examples for both types of exceptions!

Solution:
Exceptions as caused by a division-by-0 are fatal and cannot be repaired by the OS.

Similarly, the execution of privileged instructions in user-mode usually leads to the abortion of the task. However, in virtual machine environments, guest operating systems run in user-mode but "think" they were in supervisor-mode. If the guest OS tries to execute a privileged instruction, the virtual machine monitor handles the exception by applying the intended effect on the virtual hardware and lets the guest OS continue.

Pagefaults also come in two flavors: If the pagefault occurred in a region of (virtual) memory that the user program may legally access, the OS resolves the pagefault by installing physical memory to back the virtual memory region and lets the user program re-execute the memory access. If the address accessed lies in a forbidden region (e.g., many systems reserve the top 1 GB in each virtual address space for the kernel), the program is aborted as a consequence of a pagefault in this area.

Question 1.3: System Calls

a. Explain how a `trap` instruction is related to system calls.

Solution:
The `trap` instruction switches the processor to privileged mode and continues code execution at a specific location (the exact details are hardware-dependent). Consequently, the `trap` instruction can be seen as a low-level mechanism that allows to implement a higher-level mechanism, namely system calls. Other methods to synchronously invoke the kernel are also possible, for example, one can execute an invalid instruction that causes an exception and traps into the kernel. However, implementing system calls using, say, a division by zero, is probably a little odd ;-)

System calls and function calls are completely different when it comes to the mechanism their are accomplished with. Function calls are jumps or branches to an arbitrary location, while system calls are a branch to a specific, system-predefined location. Function calls remain in the current context (i.e., in the user-space process, or in the kernel) while system calls cross the boundary from user-space into the kernel.

On the x86 architecture, system calls have traditionally been implemented with `trap` instructions called `software interrupts` which, in fact, do much more than what is required for system calls. Intel and AMD have added faster, more reduced instructions for system calls, called `sysenter` and `syscall`, respectively, which avoid much of the unnecessary overhead of former `software interrupts`.

As a result, an OS today may run on old systems that support only `software interrupts`, or on new systems that support either `syscall` or `sysenter` (AMD or Intel) in addition to the old mechanism. To achieve user-level programs that are compatible with all three scenarios and still use the fastest mechanism available in each case, the actual system call mechanism has been separated from the program binaries. Instead, it is placed in a *separate code segment called a `trampoline` which is provided by the OS kernel. That way, only the kernel has to detect (typically at system boot time) the fastest usable system call mechanism and thereupon choose the appropriate `trampoline`. A user-level application now performs a function call into the `trampoline` (mapped into its address space at a known location) to invoke a system call, instead of issuing whichever `trap` instruction by itself.*

b. What is a system call number and how does it relate to the operating system API?

Solution:
When invoking a system call, a user-level application cannot directly call the function in the kernel that implements the requested service (unlike with function calls inside an application). Instead, a system call enters the kernel at a (single) predefined location (a single function). So, another means is required to specify which *system call an application wants to invoke: The `system call number`, specified as a parameter.*

All available system calls are assigned consecutive numbers (starting with zero). The system call interface in the kernel maintains a table of function pointers, indexed by the system call number. *When a user application invokes a system call, the generic system call interface evaluates the parameter that contains the `system call number`, calls the appropriate system call, and finally returns a status code and any return values back to the user.*

c. How can you pass parameters to the kernel when performing a system call?

Solution:
There are three main categories for passing parameters:

- *Registers*
- *Stack*
- *Main Memory + Location in Register*

Linux passes up to 6 parameters via registers. If more parameters need to be passed, all parameters are stored in main memory and a single register is used to hold a pointer to this memory region. Currently, there is no such system call in Linux (with more than 6 parameters). Historically, Linux was only capable of passing 4, later 5 parameters via registers; then select() and mmap() fell into the memory region category. Please note the system call number is generally the first parameter.

In Windows for x86, all system call parameters are passed via the user space stack. Upon entering the system service dispatch routine in kernel mode, eax must contain the system call number and edx must hold the user space stack pointer–the address of the system call arguments. The kernel then checks for the requested system call, how many parameters are to be expected and copies the arguments from the user stack to the kernel stack. These operations of course, include numerous sanity checks.

In Windows for x86-64, eax supplies the system call number and r10, rdx, r8, and r9 supply the first four system call arguments.

d. How are library calls related to system calls?

Solution:
All specifics, including details on how system calls are actually triggered, are typically hidden from the programmer behind an API. A programmer does not need to know how the OS implements the system calls he uses, as long as he obeys the API and knows what effects the system calls will have. Functions that invoke system calls are usually available in a run-time support library (a set of functions built into libraries included with the compiler), so system calls are available to programmers as regular function calls into that library.

In C (Linux), this interface is implemented in the glibc (unistd.h). The wrapper functions in this library take care of setting up the parameters and perform the appropriate trap. In Windows, ntdll.dll contains the wrapper functions for the system calls into the NT kernel. The Microsoft C library uses the Win32 API, which itself is built on the interface of ntdll.dll.

e. System calls enable the transition from user level to kernel level. Why do we have to be very careful when designing this transition?

Solution:
The parameters of a system call are specified by user-level software (applications). Assuming that many applications are buggy or malicious, the kernel (not the glibc/ntdll.dll interface) must validate these parameters before using them. For example, the kernel must make sure that the address of a buffer to read data into is in the application's address space. If it went unchecked, the application could trick the kernel into overwriting its own critical data structures, kernel code, or other applications' code or data.

f. What problem exists when a system call expects a pointer to a user-level buffer to which the kernel has to write data? Is there also a problem when the user-level buffer is only read?

Solution:
If the kernel receives a pointer to a user-level buffer to which it must write data, consistency checks are required to ensure that the passed pointer points to an address in the user area. If such a check is omitted, a user-level program could invoke the system call with a pointer to some memory location within the kernel area. If the kernel tries to write to that "buffer", it might overwrite some of its own data structures. This can lead to a system crash, or, even more severe, can be used by an attacker to compromise the system. If the buffer is only read, no kernel data can be corrupted. It might however be possible to disclose "secret" information — that is, kernel-internal data. For example, consider a system call that writes data provided by a user-level buffer to a file. If no consistency checks are performed and a user passes a pointer to some memory location within the kernel area, the kernel will save kernel-internal data to a file.