Operating Systems – sprint 2022
Tutorial-Assignment 1

Instructor: Hans P. Reiser

Note: You can find some introductory slides to the C programming language (and video) in the lecture's canvas course along the regular course materials. Please take a lot at that before working on the second part this tutorial (Q1.4 and following).

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 operating system 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. 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.

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 ;-)

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.

Question 1.4: Simple C program

For programming assignments, we provide templates that you have to extend with functionality according to the respective question. You can download the templates for programming assignments from Canvas. In this tutorial we use a template with a structure similar to what you will use for the programming homework assignments.

a. Acquire the template *tut01-template* for this programming question from Canvas and unpack it. Can you explain what the different files are good for? Which files are header files, and what are they used for?

Solution:
Functions are typically defined once and called from several other locations, potentially in other source files. A function should always be declared, before it is used (called or referenced otherwise) – if you omit the declaration, the compiler cannot perform checks such as matching the function's signature against a call to that function. These checks help to avoid a vast amount of silly bugs by exposing them automatically at compile-time.

Adding the necessary function declarations to each source file manually would cause much effort and the result would be terrible to maintain. A much better way is to use header files for this purpose: They serve to separate function declarations (in `.h` files) from function definitions (in `.c` files). That way, you need to write down (and change) the declarations only once (in one header file) and refer to that file (with `#include` statements) wherever you need the declarations.

In the template for this question, the header file `countchr.h` is used to declare the function `greet`. The source file `countchr.c` contains the definition of this function. The main source file `main.c` #includes *the header file `countchr.h`, so that the function is declared and the compiler can perform checks on the function call.*

You might notice the `#ifndef`, `#define`, and `#endif` preprocessor statements framing the header file. This structure ensures that the declarations get included at most once, even when the same header file is included several times (e.g., recursively by other header files). In more complex software projects, it avoids endless loops of header files mutually `#include`ing each other.

b. Implement the function `countchr` in *countchr.c*. It returns the number of occurrences of a character in an ASCII string, both supplied by the caller. The string is represented as a pointer to a contiguous sequence of `char`s in memory. The string is terminated with a null (0) character.

Solution:
Your implementation should look like this:

```
int countchr(char *string, char c) {
    int count = 0;

    while (*string != '\0') {
        if (*string == c) {
            count++;
        }

        string++;
    }

    return count;
}
```

To get the character pointed to by the given `string`-pointer, we have to dereference it. A while loop iterates over all characters in the string by incrementing the pointer until it finds the null character.

Note that incrementing `string` does not affect the pointer variable that the parent function used as argument in the call to `countchr`.

```
char *str = "Hallo␣World!";
countchr(str, 'o'); // str not affected!
```

Question 1.5: Pointers and Structs

a. Explain the concept of pointers. What do the `&` and `*` operators do when working with pointers?

Solution:
A pointer is a data type that is pointing at a value. The pointer variable itself holds the address of the value in memory. We can give a pointer a type which refers to the type of data stored at the address. `void` *denotes the absence of type. This is useful if the type of data that a pointer should point to is not known (e.g., a generic memory allocation function will return a void pointer). To define a pointer variable we precede its name with an asterisk (*).*

```
int *p; // a pointer to an integer value;
```

The `&`*-operator returns the address of a variable:*

```
int a = 5;
int *p = &a; // p holds the memory address of the variable a
```

To access the memory location pointed at by a pointer, we can dereference the pointer with the `*`*-operator:*

```
int a = 5;
int *p = &a; // p holds the memory address of the variable a
int b = *p;  // dereference p, b = 5;
```

b. What are the types of the following variables. Why can this code formatting be misleading?

```
int* a, b;
```

Solution:
a is a pointer to an integer and b is an integer. The position of the star can be misleading, because its scope seems to include b, but it does not.

c. Consider an array of `int`s in memory. How can you use the following pointers to access the fourth element? Both point at the beginning of the array.

```
int *ip;
void *vp;
```

Solution:
Since `ip` *is of the correct type, we can use array notation or pointer arithmetic and dereferencing to access the fourth integer.*

```
int a = ip[3];
int a = *(ip + 3);
```

`vp` *is an untyped pointer. Before we can perform pointer arithmetic we have to cast it to the correct type:*

```
int a = *((int*)vp + 3);
```