Operating systems. Lecture 7

Michał Goliński

2018-11-13

Introduction

Recall

- History of C/C++
- Compiler on the command line
- Automating builds with `make`

Plan for today

- CPU protection rings
- system calls
- pointers
- simplest system calls
- reading and writing files

CPU protection rings

Privilege levels

Modern CPU have a built-in security feature: rings of execution. In practice only two rings are used: for privileged and non-privileged code, even though x86 hardware is capable of some intermediate security levels.

Ring 0

The privileged code runs in ring 0. The code running in this mode can do virtually anything: communicate with hardware, manipulate page tables, change memory contents and is not supervised in any way. Of course it would be disastrous if rouge code was able to run at this level – normally only the kernel code runs in ring 0. A bug at this level can bring the system down.

Rings 1 and 2

These are the unused rings available on the Intel x86 platform. They were designed for drivers and should be considered privileged (in particular page tables can be manipulated). Major operating system do not use these rings of execution.

Ring 3

This is the ring for all the programs in the userspace. All the programs run in this ring, even those that are called normally *privileged*, i.e., are run by *root* (administrator). In principle a bug in ring 3 code should not be fatal for the system.

Security

Hardware ensures that not all operations can be carried out while in ring 3. Some very special instructions are reserved for ring 0, but more importantly, each memory page can be marked as "privileged code only". Trying to read and write from protected memory leads to a **segmentation fault**. This error usually terminates the program, after maybe creating a *core dump*.

System calls

Rationale

Switching from ring 3 to ring 0 cannot be initiated by unprivileged code (this would be a big security hole) which leads to a chicken and egg problem: user programs want to be able access hardware etc., but at the same time they have no authority to do so. The solution are the so-called *system calls*, i.e., functions carried out be the kernel on behalf of the user process.

Implementation

Interrupts are used for calling the system calls. An *interrupt table* is created in memory when a system boots. A user programs can just pick a number of a routine from this table and run it. This gives security: interrupt handlers can be changed only by the kernel code. Interrupt handlers **can** switch mode from ring 3 to ring 0. As this mechanism is based on machine code instruction its is language-agnostic.

Application binary interface

The specific details on how the arguments are passed to the system calls and which numbers correspond to which system calls comprise the platform ABI. For example they differ between 32-bit and 64-bit Linux systems, even though the set of available system calls is roughly the same. Linux kernel developers try to make sure that the system calls interface is not changed: user code created for a very old version of Linux should be able to run with recompiling on a modern system provided the necessary libraries are provided. System call are added but older ones are not removed, ever.

Application binary interface cont.

A glanco on ABI details for different platforms can be found in `man syscall`.

ABI gives stability on machine code level.

Application programming interface

API is the language specific set of subroutines, their parameters and functionalities as well as communication protocols. It differs from language to language in scope and stability. API is important for programmers and documented in the manual.

Application programming interface cont.

Some functions in the, say, C standard library have to be mapped to specific system calls available on the platform and the library will need to be rewritten to work on a different platform. At the same time the final program may need only to be recompiled or can be even run without changes (e.g., Java, Python).

ABI gives stability on source code level.

POSIX

The set of available system calls differs heavily between operating systems. On Unix/Linux a common denominator is given by the POSIX standard which extends the standard C library. Some of the functions are mapped one-to-one to Linux system calls, but some are not. Going outside POSIX may bring additional functionalities at the cost of portability.

POSIX does not cover GUI programming.

Win32

The API for windows is called Win32 (and Win64). Functionality overlaps with POSIX (but functions have different names, meaning and conventions), but Win32 is bigger and covers GUI programming.

No stable ABI nor API in the Linux kernel

Even though Linux API and ABI is stable for user processes the interfaces can change/evolve for kernel drivers. Drivers that are a part of the kernel are change by kernel maintainers, but if a company, e.g., Nvidia, decides to publish a driver without including it in the kernel, the driver has to be (partially) rewritten with each kernel release.

The Windows driver interface seems to be much more stable (across Vista, 7, 8, 8.1, 10).

Pointers

Introduction

The C language allows to pass not only data *per se* but also through *pointers*. Pointers are just memory addresses that, well, point to other interesting data. Pointers can be typed: this gives the compiler an idea about the type of object that is being stored at the address. Pointers themselves are all of the same size (64-bit on 64-bit processors).

Applications

Pointers have two applications:

- passing large amounts of data with minimal copying
- giving a function a place to return/write some data

Syntax

A typed pointer for a given type is denoted by adding a `*` to the type name. A pointer can be *dereferenced* by adding a `*` to the pointer. A variable address (pointer to a variable) can be obtained by the address-of operator (&).

```
#include <stdio.h>
int main() {
    int i = 5;
    int *p = &i;
    *p = 6;
    printf("%d\n", i);
    return 0;
}
```

`scanf`

```
#include <stdio.h>
int main() {
    int i;
    double d;
    char c;
    scanff("%d %lf %c\n", &i, &d, &c);
    return 0;
}
```

`char*`

By convention strings in C are stored as sequences of characters ending with the zero byte. One then only needs to give the start of the sequence and this is usually done by a pointer. In C strings are usually passed by `char*` pointers.

`void*`

If we need to store an address that does not contain any specific data type, we can use `void*`. It denotes a pointer without assigned type, but it can be cast to any other pointer type if needed.

Multi-level pointers

In principle one can have pointers to pointers, pointer to pointers to pointers etc. This gets very complicated quickly. Three stars are already seldom used.

Arrays

Arrays **are** implemented with pointers. The array is (almost) a pointer to the first (zero-th) element. Compiler can calculate the address of subsequent elements because it knows how many bytes one element of the array occupies. `void*` array elements are assumed to occupy one byte.

Pointer arithmetic

One can add integer values to pointers – this is equivalent to taking the next value in an array if the pointer would be in fact array. In fact `t[3]` is equivalent to `*(t+3)`. For this reason one can also write `3[t]`, which seems counterintuitive (do not write this way).

Managing files in POSIX

File descriptors

System calls working with files use *file descriptors*: small integer values which identify an open file on subsequent calls. By convention a freshly run process has three open descriptors:

- `0` – standard input
- `1` – standard output
- `2` – standard error output

Documentation

A detailed documentation on the functions can be found in the sections 2 and 3 of the manual. Usually the section 2 gives a description of the Linux-specific version and section 3 of the more general POSIX version.

Opening/creating files

To open a file one uses the `open` function:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Parameters

The `pathname` is the name of the file to be opened while `flags` is a logical combination of several options (exactly one of the first three is mandatory):

- `O_RDONLY` – file is opened in read mode
- `O_WRONLY` – file is opened in write mode
- `O_RDWR` – file is opened in read/write mode
- `O_APPEND` – file is opened in append mode
- `O_CREAT` – file is created if it does not exist
- `O_TRUNC` – truncate the file if it exists

Parameters cont.

The `mode` parameter describes the permissions of the newly created file. It is a logical combination of several options that can be found in the manual.

Return value

On success the functions returns the file descriptor for the newly open file. On failure they return -1 and `errno` is set to indicate the failure reason.

Reading from a file

To read from an open file one can use the `read` function:

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

The parameter `fd` is a descriptor of an open file, `buf` is a pointer to a buffer and `count` is the maximum number of bytes to read.

Return value

On success the functions returns the number of bytes that have been read to the buffer (can be zero!). On failure it returns -1 and `errno` is set.

Writing to a file

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

The parameter `fd` is a descriptor of an open file, `buf` is a pointer to a buffer and `count` is the number of bytes to write.

Return value

On success the functions returns the number of bytes that have been written (can be zero!). On failure it returns -1 and `errno` is set.

Moving the file offset

An open file has an associated *file offset* – the place from where data is read and where it is written, something like a caret in a text editor. `read` and `write` move it around, but it can be moved on its own with the `lseek` function:

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Parameters and return value

The first parameter is a descriptor of an opened file. The second is a number and the third is a flag how the number should be used (absolute value, relative to the current spot or relative to the end of file). The function returns the position of the offset after the change or -1 on failure and sets `errno` accordingly.

Closing a file

To close an open file one uses the `close` function:

```
#include <unistd.h>

int close(int fd);
```

The function returns 0 on success and -1 on failure and `errno` is set.

`errno`

Mechanism

Many functions need a way to give a detailed reason of why the call failed. This is sometimes very difficult to fir in the normal return values etc. For this reason the `errno` mechanism is used. `errno` is a variable (defined in the library) that is set to an apropriate error number. The meaning of specific errors can be found in the documentation. For system calls (as opposed to other library calls) this is very detailed.

Usage

To use this mechanism one should add the `errno.h` header (`cerrno` in C++). There are more than 100 different errors. It would be very tedious to check all of them. The function `perror` displays a message suitable for a specific error that occurred.

```
#include <unistd.h>
#include <errno.h>
int main() {
   close(3);
   perror(NULL);
   return 0;
}
```

Example

Copy a file

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


int main(int argc, char* argv[]) {
    int src = open(argv[1], O_RDONLY);
    if (src == -1) {
        perror(NULL);
        exit(1);
    }
```

```
    int dst = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC,
    ↪  S_IRWXU);
    if (dst == -1) {
        perror(NULL);
        exit(1);
    }

    int BUF_SIZE = 5;

    off_t end = lseek(src, 0, SEEK_END);
    printf("The source file seems to have %lld bytes.\n", (long
↪   long) end);
    lseek(src, 0, SEEK_SET);

    char buf[BUF_SIZE];
    ssize_t s;
    ssize_t total = 0;
    while (s = read(src, buf, BUF_SIZE)) {
        if (s == -1) {
            perror(NULL);
            exit(1);
        }
        if (write(dst, buf, s) == -1) {
            perror(NULL);
            exit(1);
        }
        total += s;
    }
    printf("%zd bytes were copied.\n", total);
    close(src);
    close(dst);
    return 0;
}
```