

# C++ Systems Programming on Linux

# C++ Systems Programming on Linux

Until now, most topics were about *standard* C++. The standard does not contain everything that is useful for good systems programming, such as:

- Creating, removing, renaming files and directories
- Efficient reading and writing of files
- Direct manual memory allocation from the kernel
- Networking
- Management of processes and threads

The Linux kernel in particular has a very extensive user-space C-API that can be used to directly communicate with the kernel for all of those tasks.

# POSIX and Linux API

POSIX is a standard that defines a C-API to communicate with the operating system.

- The POSIX API is supported by most Unix-like operating systems (e.g. Linux, Mac OS X)
- It is a pure C-API but can also be used directly in C++
- Consists of types, functions and constants defined in `<unistd.h>`, `<fcntl.h>`, various `<sys/*h>` files, and more

Linux defines additional types, functions and constants for Linux-specific operations that are not defined by the standard.

- Documentation of the POSIX functions can be found in man pages (usually in section 3posix or 3p)
- Linux-specific functions are also documented in man pages (usually in section 2)

# File Descriptors

A very central concept in the POSIX API are so called *file descriptors* (fds).

- File descriptors have the type `int`
- They are used as a “handle” to:
  - Files in the filesystem
  - Directories in the filesystem
  - Network sockets
  - Many other kernel objects
- Usually, fds are created by a function (e.g. `open()`) and must be closed by another function (e.g. `close()`)
- When working with fds in C++, the RAII pattern can be very useful

# Opening and Creating Files (1)

To open and create files the `open()` function can be used. It must be included from `<sys/stat.h>` and `<fcntl.h>`.

- `int open(const char* path, int flags, mode_t mode)`
- Opens the file at `path` with the given `flags` and returns an `fd` for that file
- If an error occurs, `-1` is returned
- The third argument `mode` is optional and only required when a file is created
- `flags` is a bitmap (created with bitwise or) that must contain exactly one of the following flags:
  - `O_RDONLY` Open the file only for reading.
  - `O_RDWR` Open the file for reading and writing.
  - `O_WRONLY` Open the file only for writing.
- `close()` must be used to close the `fd` returned by `open()` → RAII

## Opening and Creating Files (2)

There are more flags that can be combined with bitwise or:

- `O_CREAT` If the file does not exist, it is created with the permission bits taken from the mode argument
- `O_EXCL` Can only be used in combination with `O_CREAT`. Causes `open()` to fail and return an error when the file exists.
- `O_TRUNC` If the file exists and it is opened for writing, *truncate* the file, i.e. remove all its contents and set its length to 0.

Example:

```
#include <fcntl.h>
#include <sys/stat.h>
int main() {
    int fd = open("/tmp/testfile", O_WRONLY | O_CREAT, 0600);
    if (fd < 0) { /* error */ }
    else { close(fd); }
}
```

# Reading and Writing from Files

To read from and write to files, `read()` and `write()` from the header `<unistd.h>` can be used.

- `ssize_t read(int fd, void* buf, size_t count)`
- `ssize_t write(int fd, const void* buf, size_t count)`
- `fd` must be a valid file descriptor
- `buf` must be a memory buffer which has a size of at least `count` bytes
- The return value indicates how many bytes were actually read or written (can be up to `count`)
- Both functions return `-1` when an error occurs
- Note: Both functions may wait until data can actually be read or written which can lead to deadlocks!

# File Positions and Seeking (1)

For an opened file the kernel remembers the current position in the file.

- `read()` and `write()` start reading or writing from the current position
- They both advance the current position by the number of bytes read or written

The function `lseek()` (headers `<sys/types.h>` and `<unistd.h>`) can be used to get or set the current position.

- `off_t lseek(int fd, off_t offset, int whence)`
- `off_t` is a signed integer type
- The current position is changed according to `offset` and `whence`, which is one of the following:
  - `SEEK_SET` The current position is set to `offset`
  - `SEEK_CUR` `offset` is added to the current position
  - `SEEK_END` The current position is set to the end of the file plus `offset`
- `lseek()` returns the value of the new position, or `-1` if an error occurred



## File Positions and Seeking (2)

Example:

```
int fd = open("/etc/passwd", O_RDWR);
auto fileSize = lseek(fd, 0, SEEK_END);
lseek(fd, -4, SEEK_CUR);
write(fd, "test", 4); // overwrite the last 4 bytes
```

Note: The current position is shared between all threads. Generally, `read()`, `write()`, and `lseek()` should not be used concurrently on the same `fd`.

## Reading and Writing at Specific Offsets

There also exist two functions that read or write from a file without using the current position: `pread()` and `pwrite()` from the header `<unistd.h>`.

- `ssize_t` `pread(int fd, void* buf, size_t count, off_t offset)`
- `ssize_t` `pwrite(int fd, const void* buf, size_t count, off_t offset)`
- Conceptually, those functions work like `lseek(fd, offset, SEEK_SET)` followed by `read()` or `write()`
- However, they do not modify the current position in the file
- Should be used when reading from and writing to files from multiple threads

## Getting Metadata of Files

Meta data of files, such as the type of a file, its size, its owner, or the date it was last modified, can be read with `stat()` or `fstat()`. Required headers: `<sys/types.h>`, `<sys/stat.h>`, `<unistd.h>`.

- `int stat(const char* filename, struct stat* statbuf)`
- `int fstat(int fd, struct stat* statbuf)`
- The meta data of the file specified by `filename` or `fd` is written into `statbuf`
- Returns 0 on success, -1 on error
- `struct stat` has several member variables:
  - `mode_t st_mode` The file mode (`S_IFREG` for regular file, `S_IFDIR` for directory, `S_IFLNK` for symbolic link, ...)
  - `uid_t st_uid` The user id of the owner
  - `off_t st_size` The total size in bytes
  - ...

## Changing the Size of a File

Files can be resized by using the functions `truncate()` or `ftruncate()` from the headers `<sys/types.h>` and `<unistd.h>`.

- `int truncate(const char* path, off_t length)`
- `int ftruncate(int fd, off_t length)`
- Sets the size of the file specified by `path` or `fd` to `length` bytes
- If the new length is larger than the old, zero bytes are appended at the end
- Returns `0` on success, `-1` on error
- These functions are especially useful when files are used as a memory buffers, e.g. for a buffer manager of a database system

## More File Functions

POSIX and Linux have many more functions that deal with files and directories:

<code>mkdir()</code>	Create a directory
<code>mkdirat()</code>	Create a subdirectory in a specific directory
<code>openat()</code>	Open a file in a specific directory
<code>unlink()</code>	Remove a file
<code>unlinkat()</code>	Remove a file from a specific directory
<code>rmdir()</code>	Remove an empty directory
<code>chmod()/fchmod()</code>	Change the permissions of a file
<code>chown()/fchown()</code>	Change the owner of a file
<code>fsync()</code>	Force changes to a file to be written

...

# Memory Mapping

POSIX defines the function `mmap()` in the header `<sys/mman.h>` which can be used to manage the virtual address space of a process.

- `void*` `mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset)`
- Arguments have different meaning depending on flags
- On error, the special value `MAP_FAILED` is returned
- Always: If a pointer is returned successfully, it must be freed with `munmap()`
- `int` `munmap(void* addr, size_t length)`
- `addr` must be a value returned from `mmap()`
- `length` must be the same value passed to `mmap()`
- RAII should be used to ensure that `munmap()` is called

# Memory Mapping Files (1)

One use case for `mmap()` is to map the contents of a file into the virtual memory. To map a file, the arguments are used as follows:

- `addr`: hint for the kernel which address to use, should be `nullptr`
- `length`: length of the returned memory mapping (usually multiple of page size)
- `prot`: determines how the mapped pages may be accessed and is a combination (with bitwise or) of the following flags:
  - `PROT_EXEC` pages may be executed
  - `PROT_READ` pages may be read
  - `PROT_WRITE` pages may be written
  - `PROT_NONE` pages may not be accessed
- `flags`: should be either `MAP_SHARED` (changes to the mapped memory are written to the file) or `MAP_PRIVATE` (changes are not written to the file)
- `fd`: descriptor of an opened file
- `offset`: Offset into the file where the mapping should start (multiple of page size)

## Memory Mapping Files (2)

Example of reading integers from file /tmp/ints:

```
int fd = open("/tmp/ints", O_RDONLY);
void* mappedFile = mmap(nullptr, 4096, PROT_READ, MAP_SHARED, fd, 0);
int* fileInts = static_cast<int*>(mappedFile);
for (int i = 0; i < 1024; ++i)
    std::cout << fileInts[i] << std::endl;
munmap(mappedFile, 4096);
close(fd);
```

- Note: This assumes that integers are written in binary format to the file!
- Using `mmap()` to read from large files is often faster than using `read()`
- This is because with `mmap()` data is directly read from and written to the file without copying it to a buffer first



## Using mmap for Memory Allocation

`mmap()` can also be used to allocate memory by not associating it with a file.

- flags must be `MAP_PRIVATE | MAP_ANONYMOUS`
- fd must be `-1`
- offset must be `0`
- Other arguments have the same meaning
- Used by `malloc()` internally
- Should be used manually only to allocate very large regions of memory (at least several MBs)

Example of allocating 100 MiB of memory:

```
void* mem = mmap(nullptr, 100 * (1ull << 20),
                 PROT_READ | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS,
                 -1, 0);

// [...]
munmap(mem, 100 * (1ull << 20));
```

## Creating Processes with fork

The most common way to start a new process in Linux is using `fork()` from the headers `<sys/types.h>` and `<unistd.h>`.

- `pid_t fork()`
- When `fork()` is called, the process is duplicated (including its virtual memory with all memory mappings, open file descriptors, etc.)
- In the original process, `fork()` returns the process id of the new process, or `-1` if an error occurred
- In the new process, `fork()` returns `0`

```
std::cout << "start ";
if (fork() == 0) {
    std::cout << "new ";
} else {
    std::cout << "old ";
}
std::cout << "end ";
```

One possible output for this example is: start old end new end

## Fine-Grained Process Creation with `clone`

For greater control over creating a process, `clone()` from `<sched.h>` (which is also used by `fork()` internally) should be used.

- `int clone(int (*fn)(void*), void* child_stack, int flags, void* arg)`
- Takes a function pointer that will be executed in the new process, the new stack pointer for the process, flags, and an argument that will be passed to the function
- Returns the process id of the new process
- `flags` is 0 or a bitwise or combination of the following:
  - `CLONE_FILES` File descriptors are shared between old and new process
  - `CLONE_FS` File system information is shared (e.g. the current directory)
  - `CLONE_VM` Virtual memory is shared
  - `CLONE_PARENT` The parent process of the new process will be the parent of the current process
  - `CLONE_THREAD` The new process will be a thread in the current process
  - ...

## Executing Other Programs

To execute an entirely new program, `execve()` from `<unistd.h>` can be used.

- `int execve(const char* pathname, char* const argv[], char* const envp[])`
- `pathname` is the path to binary that should be executed
- `argv` is a pointer to a null-terminated array for the program arguments
- `envp` is a pointer to a null-terminated array for the environment variables
- On success, the new program is executed, so the function does not return
- On error, returns `-1`
- `execve()` replaces the virtual memory of the old program by the new, but it keeps all fds
- Is often used in combination with `fork()`

```
std::vector<const char*> args = {"/bin/ls", "/", nullptr};
std::vector<const char*> env = {"FOO=bar", nullptr};
if (fork() == 0) {
    execve("/bin/ls", args.data(), env.data());
}
```

# Thread Pinning

Threads can control on which physical CPU cores they run by using `sched_setaffinity()` from `<sched.h>`.



- `int sched_setaffinity(pid_t pid, size_t cpusetsize, const cpu_set_t* mask)`
- `pid` stands for the process id whose affinity should be set, or `0` which stands for the current thread
- `cpusetsize` must be set to `sizeof(cpu_set_t)`
- `mask` is a pointer to a `cpu_set_t` which describes which CPU cores the thread is allowed to run on
- Returns `0` on success, `-1` on error
- Variables of type `cpu_set_t` can be modified with `CPU_ZERO(cpu_set_t* set)` and `CPU_SET(int cpu, cpu_set_t* set)`

```
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(0, &set); CPU_SET(4, &set);
sched_setaffinity(0, sizeof(cpu_set_t), &set);
```

# Signals

In POSIX systems like Linux, every process can receive *signals*.

- Signals can either be generated by hardware (e.g. on memory access violations) or by software (by using `kill()`)
- By default, a process is either terminated or does nothing when it receives a signal
- A process can set a *signal handler* function which will be called when a signal is received
- The most common signals are:

Signal	Default	Description
SIGSEGV	terminate	"segfault", invalid memory access
SIGINT	terminate	interrupt from user, usually by pressing  + 
SIGTERM	terminate	process is terminated
SIGKILL	terminate	process is killed (cannot be caught with a signal handler)
SIGCHLD	ignore	a child process terminated

# Setting Signal Handlers (1)

Signal handlers can be set by using `sigaction()` from the header `<signal.h>`.

- `int sigaction(int signum, const struct sigaction* act, struct sigaction* sigact)`
- `signum` is the signal whose signal handler should be changed
- `act` is a pointer to the signal handler that should be set, or `nullptr` if an existing signal handler should be removed
- If `sigact` is not `nullptr`, it will contain the old signal handler after the function returns
- Returns `0` on success, `-1` on error
- `struct sigaction` has several members, the most important one is:  
`void (*sa_handler)(int)`
- `sa_handler` is a function pointer that points to the signal handler function that takes the signal as only argument

## Setting Signal Handlers (2)

As signal handlers can be called at any time while other code is running, they should avoid to interfere with memory that is currently accessed.

```
void handler(int /*signal*/) {
    std::cout << "Ctrl-C was pressed\n";
    std::exit(1);
}
struct sigaction s{}; // Use {} here to zero-initialize
s.sa_handler = handler;
sigaction(SIGINT, &s, nullptr);
```



## Sending Signals

A process can send a signal to itself or other process by using `kill()` from the headers `<sys/types.h>` and `<signal.h>`.

- `int kill(pid_t pid, int sig)`
- `pid` is the process id of the process that should receive the signal
- If `pid` is `0`, the signal is sent to all processes in the process group (i.e. to all other threads)
- If `pid` is `-1`, the signal is sent to *all* processes for which the calling process has the permission
- Returns `0` on success, `-1` on error
- With the signals `SIGUSR1` and `SIGUSR2` (“user-defined signals”) this can be used for (limited) communication between processes

## Inter-Process Communication with Pipes (1)

Using basic signals is often not sufficient for communication between processes. `pipe()` (from `<unistd.h>`) can be used instead which creates two fds that are connected to each other.

- `int pipe(int pipefd[2])`
- Takes a pointer to an array that can hold two integers
- Returns 0 on success, -1 on error
- Creates a unidirectional connection between `pipefd[0]` and `pipefd[1]`
- Everything that is written to `pipefd[1]` can be read from `pipefd[0]`
- Both fds must be closed eventually

```
int fds[2];
pipe(fds);
int readfd = fds[0]; int writefd = fds[1];
write(writefd, "hello", 5);
char buffer[5];
read(readfd, buffer, 5); // buffer now contains "hello"
close(readfd); close(writefd);
```

## Inter-Process Communication with Pipes (2)

pipe() is usually used in combination with fork():

```
int fds[2]; pipe(fds);
int readfd = fds[0];
int writefd = fds[1];
if (fork() == 0) {
    // We only need to read from the parent, so close writefd
    close(writefd);
    char buffer[6]; buffer[5] = 0;
    read(readfd, buffer, 5);
    std::cout << "parent wrote: " << buffer;
    close(readfd)
} else {
    // Likewise, close readfd
    close(readfd);
    write(writefd, "hello", 5);
    close(writefd);
}
```



# Error Handling

Most functions use `errno` from the header `<cerrno>` for error handling.

- `errno` is a global variable that contains an error code
- Is set when a function returns an error (e.g. by returning `-1`)
- All possible values for `errno` are available as constants:
  - `EINVAL` Invalid argument
  - `ENOENT` No such file or directory (e.g. in `open()`)
  - `EACCES` Permission denied
  - `ENOMEM` Not enough memory (e.g. for `mmap()`)
  - ...
- A description of the error can be retrieved with `std::strerror()` from `<cstring>`