Concepts of C++ Programming

Lecture 13: I/O and Testing

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

Systems Programming in C++

- So far: mostly covered standard C++
- Standard does not contain everything required for OS interaction
  - Efficient file I/O
  - Networking
  - Direct memory allocation from the OS
  - ...
- Such operations need a different interface to the OS

POSIX and Linux API

- POSIX: Standard defining C-API ($\rightsquigarrow$ usable in C++) for OS interaction
- Supported on most Unix-like operating systems
- Defines several data types, functions, and constants (macros)
  e.g. in `unistd.h`, `fcntl.h`, `sys/*.h`
- Linux defines additional types, functions, and constants
- Documented in `man` pages, usually sections 2 and 3

File Descriptors

- File descriptor: handle to resource managed by OS
  - Files/directories in filesystem
  - Network sockets
  - Many other kernel objects
- Usually created by a function (e.g. `open`) and closed by `close`
- In C++, the RAII pattern can be very useful

Opening and Creating Files

- `int open(const char* path, int flags, mode_t mode)`
  - Argument `mode` is optional, only required when file is created
- Open file at path and return fd for that file, or -1 on error

- Flags is a bitwise combination of flags and must contain exactly one of:
  - `O_RDONLY`, `O_RDWR`, `O_WRONLY`
- Flag `O_CREAT`: create file if it doesn't exist
- Flags `O_CREAT|O_EXCL`: create file if it doesn't exist, error if it does exist
- Flag `O_TRUNC`: if file exists, truncate it (remove all content)

open Example

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
int main() {
    int fd = open("/tmp/testfile", O_WRONLY | O_CREAT, 0600);
    if (fd < 0) {
        perror("/tmp/testfile");
        return 1;
    }
    close(fd);
}
```

Reading/Writing Files

- `ssize_t read(int fd, void* buf, size_t count)`
- `ssize_t write(int fd, const void* buf, size_t count)`
- Read/write *up to* `count` bytes to/from `buf`
  - Can always read/write less than `count`
- Returns number of bytes read/written, -1 indicates error
- `read` return value 0: reached end of file

- Functions might block until data can be read/written

  ⇝ Might lead to deadlocks!

Error Handling

- Most functions use `errno` (`<cerrno>`[154]) for error handling
- `errno`: thread-local global variable containing an error code

- If a function return -1, `errno` is set to the error code
  - `EINVAL`: invalid argument
  - `ENOENT`: no such file or directory
  - `EACCESS`: permission denied
  - … (see `man 3 errno`)
- Error message can be retrieved using `std::strerror()` from `<cstring>`

[154] https://en.cppreference.com/w/cpp/header/cerrno

File Positions and Seeking

- For every file descriptor: kernel remembers position in file
- `read`/`write` start at and advance that position

- `off_t lseek(int fd, off_t offset, int whence)`
  get and/or set current position
- `whence == SEEK_SET`: set current position to `offset`
- `whence == SEEK_CUR`: add `offset` to current position
- `whence == SEEK_END`: set current position to end of file plus `offset`
- Return value: new position in file, or -1 to indicate an error

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

Reading and Writing at Specific Offset

- Current offset into file is shared between all threads
- Problematic when reading/writing in parallel

- `ssize_t pread(int fd, void* buf, size_t count, off_t off)`
- `ssize_t pwrite(int fd, const void* buf, size_t count, off_t off)`
- Read/write ad specified offset, don't modify current position

- `int ftruncate(int fd, off_t length)`
- Set size of file, if larger than previous size fill with zero bytes

Metadata of Files

- `int stat(const char* path, struct stat* statbuf)`
- `int fstat(int fd, struct stat* statbuf)`
  - `<sys/types.h>`, `<sys/stat.h>`, `<unistd.h>`
- Write metadata of specified path into `statbuf`

- `st_mode`: File type and mode (e.g., permissions)
- `st_uid`: user id of the file owner
- `st_size`: size of the file
- `st_mtime`: timestamp of last modification
- ...

UNIX File Types

- Regular files
- Directories
- Symbolic links (often implicitly followed)
- Pipes
- Character devices (e.g., terminal, `/dev/urandom`)
- Block devices (e.g., disks)
- Sockets

- `st_size` shows actual size *only* for regular files and block devices

Checking for Non-Existing Files

△ Quiz: What is NOT problematic about this code?

```
// Includes <err.h> <fcntl.h> <sys/stat.h>
struct stat statbuf;
if (stat(argv[1], &statbuf) < 0)
    err(1, "%s", argv[1]);
if (!S_ISREG(statbuf.st_mode))
    errx(1, "%s:␣Not␣a␣regular␣file", argv[1]);
int fd = open(argv[1], O_RDONLY); // No need to check error, file exists
```

A. Process might not have permission to open/read the file
B. stat doesn't follow symbolic links, but open does
C. stat might refer to a different file than open
D. open might return `EINTR`, where the function should be restarted

C++ Streams[155]

- C++ library for I/O designed arount the concepts of *streams*
- `std::istream`: base class for input operations (`operator>>`)
- `std::ostream`: base class for output operations (`operator<<`)
- `std::iostream`: subclass of `std::istream` and `std::ostream`
- `std::cin`/`std::cout`: streams for standard input/output

- Like `std::string`, actually templates parameterized for `char`

[155] https://en.cppreference.com/w/cpp/io

Input and Output Streams

- `operator>>()`: read value of given type, skip leading whitespace
- `operator<<()`: write value of given type
  - Both operators can be overloaded for own types as second argument
- `get()`/`put()`: read/write single character
- `read()`/`write()`: read/write multiple characters

```
// Defined by the standard library:
std::istream& operator>>(std::istream&, int&);
int value;
std::cin >> value;

// Write 1024 chars to cout:
std::vector<char> buffer(1024);
std::cout.write(buffer.data(), 1024);
```

Common Operations

- Various methods to check whether stream is in specific error state
- `good()`: no error occurred
- `fail()`: an error occurred
- `bad()`: a non-recoverable error occurred
- `eof()`: reached end-of-file
- `operator bool()`: true if stream has no errors

```cpp
int value;
if (std::cin >> value) {
    std::cout << "value = " << value << std::endl;
} else {
    std::cout << "error" << std::endl;
}
```

`std::endl`

△ Quiz: Which statement is correct?

A. `std::cout << std::endl` is equivalent to `std::endl(std::cout)`.
B. `std::cout << std::endl` is equivalent to `std::cout << '\n'`.
C. `std::endl` is an object type and `operator<<` has a special overload.
D. `std::endl` is more efficient than writing a new line character.

- Flushing an output stream is often not necessary
- Prefer writing newline characters instead

File Streams

- `std::ifstream`: file stream to read file
- `std::ofstream`: file stream to write file
- `std::fstream`: file stream to read an write file

```
std::ifstream input("input_file");
if (!input) { std::cout << "couldn't open input_file\n"; }
std::ofstream output("output_file");
if (!output) { std::cout << "couldn't open output_file\n"; }
// Read an int from input_file and write it to output_file
int value = -1;
if (!(input >> value)) {
    std::cout << "couldn't read from file\n";
}
if (!(output << value)) {
    std::cout << "couldn't write to file\n";
}
```

Reading a File Into Memory

```
std::string readFile(const char* path) {
  auto stream = std::ifstream(path, std::ios::in);
  stream.seekg(0, std::ios::end);
  auto size = stream.tellg();
  stream.seekg(0, std::ios::beg);
  std::vector<char> data(size);
  stream.read(&data[0], size);
  return std::string(&data[0], size);
}
```

- This is *not* how to do it

Disadvantages of Streams

- Streams make heavy use of virtual functions and virtual inheritance
- System's locale settings are respected $\rightsquigarrow$ slower
  - E.g., whether dot or comma is used for floating-point numbers
  - Especially handling of numbers is very inefficient
- Streams have implicit state (e.g., formatting specifiers, error status)
- Many important operations (e.g. `stat`) are not exposed,
  no way of accessing the underlying file descriptor

$\Rightarrow$ Avoid using C++ streams, better use OS-specific functions

I/O Performance and Buffering

- I/O operations are often slow (e.g., hard disk, network, etc.)
  - ⇒ Kernel doesn't immediately write file to disk
  - Instead, writing data is often delayed for some time
  - Buffers flushed on close or `fsync`
- System calls are somewhat slow (context switch, etc.)
  - ⇒ Standard library doesn't immediately calls kernel
  - Instead, data is buffered in user-space for some time
  - Buffers flushed on close, exit, or flush
- Techniques for more efficient I/O: `mmap`, `io_uring`, ...
  all of these are somewhat-to-very OS-specific and non-portable.

`close`

△ Quiz: What can happen when the error of `close()` is ignored?

A. Silent data loss.
B. File descriptor leak.
C. Nothing, `close` cannot return an error.

std::filesystem[156]

- C++17 addition, provides interface for working with paths and files
- Provides abstractions for several POSIX functions
  - But: not all, and often doesn't expose the required interface
- `std::filesystem::path` is useful for working with file paths
  - Convenience functions for concatenating, adding suffixes, etc.

- Cannot provide the same guarantees as OS-defined functions

[156] https://en.cppreference.com/w/cpp/filesystem

Testing

Tests should be an integral part of every larger project

- Unit tests
- Integration tests
- ...

Good test coverage greatly facilitates implementing a large project

- Tests can ensure (to some extent) that modifications do not break existing functionality
- Can easily refactor code
- Can easily change the internals of a component
- ...

Googletest (1)

- Works on a large variety of platforms
- Contains a large set of useful functions
- Can usually be installed through a package manager
- Can be added to a CMake project through the `FindGTest.cmake` module

Functionality overview

- Test cases
- Predefined and user-defined assertions
- Death tests
- ...

Googletest (2)

Simple tests

```
#include <gtest/gtest.h>
TEST(TestSuiteName, TestName) {
    ...
}
```

- Defines and names a test function that belongs to a test suite
- Test suites can for example map to one class or function
- Googletest assertions can be used to control the outcome of the test function
- If any assertion fails or the test function crashes, the entire test case fails

Googletest (3)

Fatal assertions

- Fatal assertions are prefixed with `ASSERT_`
- When a fatal assertion fails the test function is immediately terminated

Non-fatal assertions

- Non-fatal assertions are prefixed with `EXPECT_`
- When a non-fatal assertion fails the test function is allowed to continue
- Nevertheless the test case will fail
- All assertions exist in fatal and non-fatal versions

Assertion examples

- `ASSERT_TRUE(condition);` or `ASSERT_FALSE(condition);`
- `ASSERT_EQ(val1, val2);` or `ASSERT_NE(val1, val2);`
- ...

Googletest (4)

A custom `main` function needs to be provided for Googletest

```
#include <gtest/gtest.h>
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

- Should usually be placed in a separate `Tester.cpp` or `main.cpp`

Example: Average of Two Integers

- Compute the average of two integers, round toward the first

(see script)

[157] https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/

Coverage (1)

Code coverage can help ensure proper testing of a project

- Simple metrics like line coverage have to be interpreted carefully
- Can indicate that a certain part of a project has *not* been tested properly
- Can usually *not* indicate that a certain part of a project has been tested exhaustively

Line coverage information can automatically be collected during test execution

- Possible with a variety of tools
- GCC contains the built-in coverage tool `gcov`
- Clang can produce `gcov`-like output
- `lcov` together with `genhtml` can be used to generate HTML line coverage reports from information collected during test execution

Coverage (2)

Brief example

```
# build executable with gcov enabled
> g++ -fprofile-arcs -ftest-coverage -o main main.cpp

# run executable and generate coverage data
> ./main

# generate lcov report
> lcov -c --directory . --output-file coverage.info --ignore-errors mismatch

# generate html report
> genhtml coverage.info --output-directory coverage
```

- Produces HTML coverage report in `coverage/index.html`
- Configuration for coverage reports should be part of CMake configuration

Integration Tests

- Writing fine-granular unit tests can be quite tedious
- High overhead when refactoring code: need to adjust all tests

- In practice: unit tests complemented with integration tests
- For example: test I/O behavior of the entire program

FileCheck[158] Tests

- FileCheck: utility from LLVM to verify output against expectation

```c
// llvm-project/clang/test/Lexer/counter.c
// RUN: %clang -E %s | FileCheck %s

#define PASTE2(x,y) x##y
#define PASTE1(x,y) PASTE2(x,y)
#define UNIQUE(x) PASTE1(x,__COUNTER__)

A: __COUNTER__
B: UNIQUE(foo);
C: UNIQUE(foo);
D: __COUNTER__
// CHECK: A: 0
// CHECK: B: foo1;
// CHECK: C: foo2;
// CHECK: D: 3
```

[158] https://llvm.org/docs/CommandGuide/FileCheck.html

Auto-Generating Tests

- Sometimes, expected output of tests can change
- Sometimes, this is due to unrelated changes
  - E.g., when adding an optimization to a compiler, the output of other tests changes

- Adjusting all tests manually is a huge effort
- Having a tool to auto-generate the expected output reduces this
  - Only need to review code changes in `git diff`

I/O and Testing – Summary

- POSIX provides a somewhat portable and rather low-level operating system interface
  for interacting with the file system
- File I/O in POSIX centered around file descriptors
- C++ I/O designed around streams as a higher-level abstraction
- C++ streams are inefficient and limited in features
- C++ Filesystem API provides good abstraction for paths
- Unit tests and integration tests are important for quality

I/O and Testing – Questions

- When do `read`/`write` return? What does a return value 0 imply?
- What types of errors can occur during `close()`?
- How to reliably get the size of a file for reading it into memory?
- What is the difference between `bad()` and `fail()` on streams?
- What are disadvantages of streams over using OS-specific functions directly?
- How to get code coverage information from unit tests? What does this mean?
- What are benefits of integration tests over unit tests?