Министерство науки и высшего образования Российской Федерации
федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Операционные системы

Лабораторная работа №2

Группа: P3324

Выполнил: Маликов Глеб Игоревич

Проверил:

Клименков Сергей Викторович

Санкт-Петербург

2024г.

Оглавление

Задание

Вариант

- ОС: Linux
- Second-change

Для оптимизации работы с блочными устройствами в ОС существует кэш страниц с данными, которыми мы производим операции чтения и записи на диск. Такой кэш позволяет избежать высоких задержек при повторном доступе к данным, так как операция будет выполнена с данными в RAM, а не на диске (вспомним пирамиду памяти).

В данной лабораторной работе необходимо реализовать блочный кэш в пространстве пользователя в виде динамической библиотеки (dll или so). Политику вытеснения страниц и другие элементы задания необходимо получить у преподавателя.

При выполнении работы необходимо реализовать простой API для работы с файлами, предоставляющий пользователю следующие возможности:

1. Открытие файла по заданному пути файла, доступного для чтения. Процедура возвращает некоторый хэндл на файл. Пример: int lab2_open(const char* path);
2. Закрытие файла по хэндлу. Пример: int lab2_close(int fd);
3. Чтение данных из файла. Пример: ssize_t lab2_read(int fd, void buf[.count], size_t count);
4. Запись данных в файл. Пример: ssize_t lab2_write(int fd, const void buf[.count], size_t count);
5. Перестановка позиции указателя на данные файла. Достаточно поддержать только абсолютные координаты. Пример: off_t lab2_lseek(int fd, off_t offset, int whence);
6. Синхронизация данных из кэша с диском. Пример: int lab2_fsync(int fd);

Операции с диском разработанного блочного кеша должны производиться в обход page cache используемой ОС.

В рамках проверки работоспособности разработанного блочного кэша необходимо адаптировать указанную преподавателем программу-загрузчик из ЛР 1, добавив использование кэша. Запустите программу и убедитесь, что она корректно работает. Сравните производительность до и после.

Ограничения

- Программа (комплекс программ) должна быть реализован на языке C или C++.
- Если по выданному варианту задана политика вытеснения Optimal, то необходимо предоставить пользователю возможность подсказать page cache, когда будет совершен следующий доступ к данным. Это можно сделать либо добавив параметр в процедуры read и write (например, ssize_t lab2_read(int fd, void buf[.count], size_t count, access_hint_t hint)), либо добавив еще одну функцию в API (например, int lab2_advice(int fd, off_t offset, access_hint_t hint)). access_hint_t в данном случае – это абсолютное время или временной интервал, по которому разработанное API будет определять время последующего доступа к данным.
- Запрещено использовать высокоуровневые абстракции над системными вызовами. Необходимо использовать, в случае Unix, процедуры libc.

Решение

Структура проекта

```
.
├── API
│   ├── lab2_cache.cpp
│   └── lab2_cache.h
├── Bench
│   ├── IOLatencyReadBenchmark.cpp
│   ├── IOLatencyReadBenchmark.h
│   ├── IOLatencyWriteBenchmark.cpp
│   ├── IOLatencyWriteBenchmark.h
│   ├── RandReadBenchmark.cpp
│   ├── RandReadBenchmark.h
│   ├── RandWriteBenchmark.cpp
│   └── RandWriteBenchmark.h
├── BenchUtils
│   ├── BenchmarkConfig.h
│   ├── BenchmarkMain.h
│   └── BenchmarkUtils.cpp
├── CMakeLists.txt
├── io-lat-read.cpp
├── io-lat-write.cpp
├── rand-read.cpp
├── rand-write.cpp
├── README.md
├── run_bench.sh
└── run_rand_bench.sh
```

```
cmake_minimum_required(VERSION 3.29)
cmake_minimum_required(VERSION 3.29)
project(OS_Lab2)

set(CMAKE_CXX_STANDARD 20)

#set(CMAKE_CXX_FLAGS_DEBUG "-O3 -g -march=native -mtune=native -flto -
funroll-loops -fomit-frame-pointer")
set(CMAKE_CXX_FLAGS_DEBUG "-O0 -g")

# Dynamic library
add_library(lab2_cache SHARED API/lab2_cache.cpp)

# io-lat-write
add_executable(OS_Lab2_write io-lat-write.cpp
        Bench/IOLatencyWriteBenchmark.cpp
        Bench/IOLatencyWriteBenchmark.h
        BenchUtils/BenchmarkMain.h)
target_link_libraries(OS_Lab2_write lab2_cache)
target_include_directories(OS_Lab2_write PRIVATE API)

# io-lat-read
add_executable(OS_Lab2_read io-lat-read.cpp
        Bench/IOLatencyReadBenchmark.cpp
        Bench/IOLatencyReadBenchmark.h
        BenchUtils/BenchmarkMain.h)
target_link_libraries(OS_Lab2_read lab2_cache)
target_include_directories(OS_Lab2_read PRIVATE API)
```

```
# rand-read
add_executable(OS_Lab2_rand_read rand-read.cpp
        Bench/RandReadBenchmark.cpp
        Bench/RandReadBenchmark.h
        BenchUtils/BenchmarkMain.h)
target_link_libraries(OS_Lab2_rand_read lab2_cache)
target_include_directories(OS_Lab2_rand_read PRIVATE API)

# rand-write
add_executable(OS_Lab2_rand_write rand-write.cpp
        Bench/RandWriteBenchmark.cpp
        Bench/RandWriteBenchmark.h
        BenchUtils/BenchmarkMain.h)
target_link_libraries(OS_Lab2_rand_write lab2_cache)
target_include_directories(OS_Lab2_rand_write PRIVATE API)
```

Программы-загрузчики

io-lat-write

Измерение задержки на запись накопителя с размерами блока Block Size.

```
#include "IOLatencyWriteBenchmark.h"
#include <iostream>
#include <fstream>
#include <chrono>
#include <vector>
#include <numeric>
#include <algorithm>
#include <cstring>
#include <fcntl.h>
#include <unistd.h>

#include "../API/lab2_cache.h"

namespace IOLatencyWriteBenchmark {

    constexpr size_t BLOCK_SIZE = 512; // Writing in blocks of 512 bytes
    constexpr size_t FILE_SIZE = 1024 * 1024; // File size of 1 MB

    void run(const int iterations, const bool verbose, const bool
use_cache) {
        const std::vector buffer(BLOCK_SIZE, 'a'); // Buffer for writing
        std::vector<double> durations; // Time durations for each iteration
        durations.reserve(iterations);

        for (int i = 0; i < iterations; ++i) {
            remove("testfile.dat"); // Delete the file if it already exists
for clean write

            auto start = std::chrono::high_resolution_clock::now();

            if (use_cache) {
                // Using the lab2_cache API
                const int fd = lab2_open("testfile.dat", O_CREAT | O_RDWR |
O_TRUNC, 0644);
                if (fd < 0) {
                    std::cerr << "Error opening file for IO benchmark!" <<
std::endl;
                    return;
                }
```

```
                // Writing data to the file in blocks
                for (size_t written = 0; written < FILE_SIZE; written +=
BLOCK_SIZE) {
                    ssize_t ret = lab2_write(fd, buffer.data(),
BLOCK_SIZE);
                    if (ret != BLOCK_SIZE) {
                        std::cerr << "Error writing to file during IO
benchmark!" << std::endl;
                        lab2_close(fd);
                        return;
                    }
                }
                lab2_fsync(fd);
                lab2_close(fd);
            } else {
                // Use standard file IO with O_DIRECT
                const int fd = open("testfile.dat", O_CREAT | O_RDWR |
O_TRUNC | O_DIRECT, 0644);
                if (fd < 0) {
                    std::cerr << "Error opening file for IO benchmark!" <<
std::endl;
                    return;
                }

                // Allocate an aligned buffer
                void* aligned_buffer = nullptr;
                if (posix_memalign(&aligned_buffer, 512, BLOCK_SIZE) != 0)
{
                    std::cerr << "Error allocating aligned buffer!" <<
std::endl;
                    close(fd);
                    return;
                }
                std::memset(aligned_buffer, 'a', BLOCK_SIZE);

                // Write data to the file in blocks
                for (size_t written = 0; written < FILE_SIZE; written +=
BLOCK_SIZE) {
                    ssize_t ret = write(fd, aligned_buffer, BLOCK_SIZE);
                    if (ret != static_cast<ssize_t>(BLOCK_SIZE)) {
                        std::cerr << "Error writing to file during IO
benchmark!" << std::endl;
                        free(aligned_buffer);
                        close(fd);
                        return;
                    }
                }
                fsync(fd);
                free(aligned_buffer);
                close(fd);
            }

            auto end = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double> duration = end - start;
            durations.push_back(duration.count());

            if (verbose) std::cout << "IO Iteration " << i + 1
                      << ": Write latency = " << duration.count() << "
seconds"
                      << std::endl;
        }

        // Calculating overall statistics
        const double avg_duration = std::accumulate(durations.begin(),
```

```
durations.end(), 0.0) / durations.size();
        const double min_duration = *std::ranges::min_element(durations);
        const double max_duration = *std::ranges::max_element(durations);

        std::cout << "\nOverall Statistics:\n";
        std::cout << "Average write latency: " << avg_duration << "
seconds\n";
        std::cout << "Minimum write latency: " << min_duration << "
seconds\n";
        std::cout << "Maximum write latency: " << max_duration << "
seconds\n";

        if (use_cache) {
            std::cout << "Cache hits: " << lab2_get_cache_hits() <<
std::endl;
            std::cout << "Cache misses: " << lab2_get_cache_misses() <<
std::endl;
            lab2_reset_cache_counters();
        }
    }
}
```

io-lat-read

Измерение задержки на чтение накопителя с размерами блока Block Size.

```
#include "IOLatencyReadBenchmark.h"
#include <iostream>
#include <fstream>
#include <chrono>
#include <vector>
#include <numeric>
#include <algorithm>
#include <cstring>
#include <fcntl.h>
#include <unistd.h>
#include "../API/lab2_cache.h"

namespace IOLatencyReadBenchmark {

    constexpr size_t BLOCK_SIZE = 512; // Reading in blocks of 512 bytes

    void run(const std::string& file_path, const int iterations, const bool
verbose, const bool use_cache) {
        std::vector<char> buffer(BLOCK_SIZE); // Buffer for reading
        std::vector<double> durations; // Time durations for each iteration
        durations.reserve(iterations);

        for (int i = 0; i < iterations; ++i) {
            auto start = std::chrono::high_resolution_clock::now();

            if (use_cache) {
                // Using the lab2_cache API
                const int fd = lab2_open(file_path.c_str(), O_RDONLY, 0);
                if (fd < 0) {
                    std::cerr << "Error opening file for IO benchmark!" <<
std::endl;
                    return;
                }

                ssize_t bytes_read = 0;
                do {
                    bytes_read = lab2_read(fd, buffer.data(), BLOCK_SIZE);
```

```
                    if (bytes_read < 0) {
                        std::cerr << "Error reading from file during IO
benchmark!" << std::endl;
                        lab2_close(fd);
                        return;
                    }
                } while (bytes_read > 0);

                lab2_close(fd);
            } else {
                // Use standard file IO with O_DIRECT
                const int fd = open(file_path.c_str(), O_RDONLY |
O_DIRECT);
                if (fd < 0) {
                    std::cerr << "Error opening file for IO benchmark!" <<
std::endl;
                    return;
                }

                // Allocate an aligned buffer
                void* aligned_buffer = nullptr;
                if (posix_memalign(&aligned_buffer, 512, BLOCK_SIZE) != 0)
{
                    std::cerr << "Error allocating aligned buffer!" <<
std::endl;
                    close(fd);
                    return;
                }

                ssize_t bytes_read = 0;
                do {
                    bytes_read = read(fd, aligned_buffer, BLOCK_SIZE);
                    if (bytes_read < 0) {
                        std::cerr << "Error reading from file during IO
benchmark!" << std::endl;
                        free(aligned_buffer);
                        close(fd);
                        return;
                    }
                } while (bytes_read > 0);

                free(aligned_buffer);
                close(fd);
            }

            auto end = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double> duration = end - start;
            durations.push_back(duration.count());

            if (verbose) {
                std::cout << "IO Iteration " << i + 1
                          << ": Read latency = " << duration.count() << "
seconds"
                          << std::endl;
            }
        }

        const double avg_duration = std::accumulate(durations.begin(),
durations.end(), 0.0) / durations.size();
        const double min_duration = *std::ranges::min_element(durations);
        const double max_duration = *std::ranges::max_element(durations);

        std::cout << "\nOverall Statistics:\n";
        std::cout << "Average read latency: " << avg_duration << "
seconds\n";
```

```
        std::cout << "Minimum read latency: " << min_duration << "
seconds\n";
        std::cout << "Maximum read latency: " << max_duration << "
seconds\n";

        if (use_cache) {
            std::cout << "Cache hits: " << lab2_get_cache_hits() <<
std::endl;
            std::cout << "Cache misses: " << lab2_get_cache_misses() <<
std::endl;
            lab2_reset_cache_counters();
        }
    }
}
```

rand-write

Измерение задержки на запись накопителя с размерами блока Block Size. Случайная запись.

```
#include "RandWriteBenchmark.h"
#include <iostream>
#include <fstream>
#include <chrono>
#include <vector>
#include <numeric>
#include <algorithm>
#include <random>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <cstring>
#include "../API/lab2_cache.h"

namespace RandWriteBenchmark {

    constexpr size_t BLOCK_SIZE = 512;
    constexpr size_t FILE_SIZE = 1024 * 1024; // File size of 1 MB
    constexpr auto DEFAULT_FILE_PATH = "testfile.dat";

    void run(const int iterations, const bool verbose, const bool
use_cache) {
        const std::string file_path = DEFAULT_FILE_PATH;

        const std::vector buffer(BLOCK_SIZE, 'B');
        std::vector<double> durations;
        durations.reserve(iterations);

        // Get file size and ensure it's large enough
        struct stat st = {};
        if (stat(file_path.c_str(), &st) != 0) {
            std::cerr << "Failed to get file size for " << file_path << ":
" << std::strerror(errno) << std::endl;
            return;
        }
        const size_t file_size = st.st_size;

        if (file_size < BLOCK_SIZE) {
            std::cerr << "File size must be at least " << BLOCK_SIZE << "
bytes." << std::endl;
            return;
        }

        // Prepare random offset generator with block alignment
```

```
        std::random_device rd;
        std::mt19937 gen(rd());
        const off_t max_block_index = (file_size / BLOCK_SIZE) - 1;
        std::uniform_int_distribution<off_t> block_dist(0,
max_block_index);

        for (int i = 0; i < iterations; ++i) {
            const off_t random_offset = block_dist(gen) * BLOCK_SIZE;

            auto start = std::chrono::high_resolution_clock::now();

            if (use_cache) {
                const int fd = lab2_open(file_path.c_str(), O_RDWR, 0);
                if (fd < 0) {
                    std::cerr << "Error opening file '" << file_path << "'
for random write benchmark: "
                              << std::strerror(errno) << std::endl;
                    return;
                }

                if (lab2_lseek(fd, random_offset, SEEK_SET) < 0) {
                    std::cerr << "Error seeking in file '" << file_path <<
"': "
                              << std::strerror(errno) << std::endl;
                    lab2_close(fd);
                    return;
                }

                for (size_t written = 0; written < FILE_SIZE; written +=
BLOCK_SIZE) {
                    ssize_t bytes_written = lab2_write(fd, buffer.data(),
BLOCK_SIZE);
                    if (bytes_written < 0) {
                        std::cerr << "Error writing to file '" << file_path
                                  << "' during random write benchmark: " <<
std::strerror(errno) << std::endl;
                        lab2_close(fd);
                        return;
                    }
                }

                lab2_fsync(fd);
                lab2_close(fd);
            } else {
                // Standard file IO with O_DIRECT for direct writes
                const int fd = open(file_path.c_str(), O_RDWR | O_DIRECT);
                if (fd < 0) {
                    std::cerr << "Error opening file '" << file_path << "'
for random write benchmark: "
                              << std::strerror(errno) << std::endl;
                    return;
                }

                void* aligned_buffer = nullptr;
                if (posix_memalign(&aligned_buffer, 512, BLOCK_SIZE) != 0)
{
                    std::cerr << "Error allocating aligned buffer!" <<
std::endl;
                    close(fd);
                    return;
                }

                std::memset(aligned_buffer, 'B', BLOCK_SIZE);

                if (lseek(fd, random_offset, SEEK_SET) < 0) {
```

```
                    std::cerr << "Error seeking in file '" << file_path <<
"': "
                              << std::strerror(errno) << std::endl;
                    free(aligned_buffer);
                    close(fd);
                    return;
                }

                for (size_t written = 0; written < FILE_SIZE; written +=
BLOCK_SIZE) {
                    ssize_t bytes_written = write(fd, aligned_buffer,
BLOCK_SIZE);
                    if (bytes_written < 0) {
                        std::cerr << "Error writing to file '" << file_path
                                  << "' during random write benchmark: " <<
std::strerror(errno) << std::endl;
                        free(aligned_buffer);
                        close(fd);
                        return;
                    }
                }

                fsync(fd);
                free(aligned_buffer);
                close(fd);
            }

            auto end = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double> duration = end - start;
            durations.push_back(duration.count());

            if (verbose) {
                std::cout << "Random Write Iteration " << i + 1
                          << ": Latency = " << duration.count() << "
seconds"
                          << std::endl;
            }
        }

        const double avg_duration = std::accumulate(durations.begin(),
durations.end(), 0.0) / durations.size();
        const double min_duration = *std::ranges::min_element(durations);
        const double max_duration = *std::ranges::max_element(durations);

        std::cout << "=== Random Write Benchmark Results ===\n";
        std::cout << "Average write latency: " << avg_duration << "
seconds\n";
        std::cout << "Minimum write latency: " << min_duration << "
seconds\n";
        std::cout << "Maximum write latency: " << max_duration << "
seconds\n";

        if (use_cache) {
            std::cout << "Cache hits: " << lab2_get_cache_hits() <<
std::endl;
            std::cout << "Cache misses: " << lab2_get_cache_misses() <<
std::endl;
            lab2_reset_cache_counters();
        }
    }

}
```

rand-read

Измерение задержки на чтение накопителя с размерами блока Block Size. Случайное чтение.

```
#include "RandReadBenchmark.h"
#include <iostream>
#include <fstream>
#include <chrono>
#include <vector>
#include <numeric>
#include <algorithm>
#include <random>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <cstring>
#include "../API/lab2_cache.h"

namespace RandReadBenchmark {

    constexpr size_t BLOCK_SIZE = 512;

    void run(const std::string& file_path, const int iterations, const bool
verbose, const bool use_cache) {
        std::vector<char> buffer(BLOCK_SIZE);
        std::vector<double> durations;
        durations.reserve(iterations);

        struct stat st = {};
        if (stat(file_path.c_str(), &st) != 0) {
            std::cerr << "Failed to get file size: " <<
std::strerror(errno) << std::endl;
            return;
        }
        const size_t file_size = st.st_size;

        if (file_size == 0) {
            std::cerr << "File is empty. Cannot perform reads." <<
std::endl;
            return;
        }

        // Prepare random offset generator with block alignment
        std::random_device rd;
        std::mt19937 gen(rd());
        const off_t max_block_index = file_size >= BLOCK_SIZE ? (file_size
- BLOCK_SIZE) / BLOCK_SIZE : 0;
        std::uniform_int_distribution<off_t> block_dist(0,
max_block_index);

        for (int i = 0; i < iterations; ++i) {
            const off_t random_offset = block_dist(gen) * BLOCK_SIZE;

            const auto start = std::chrono::high_resolution_clock::now();

            if (use_cache) {
                const int fd = lab2_open(file_path.c_str(), O_RDONLY, 0);
                if (fd < 0) {
                    std::cerr << "Error opening file for random read
benchmark: " << std::strerror(errno) << std::endl;
                    return;
                }

                // Seek to random offset
                if (lab2_lseek(fd, random_offset, SEEK_SET) < 0) {
                    std::cerr << "Error seeking in file: " <<
```

```
std::strerror(errno) << std::endl;
                    lab2_close(fd);
                    return;
                }

                for (int j = 0; j < 2048; ++j) { // 2048 * 512 = 1MB
                    ssize_t bytes_read = lab2_read(fd, buffer.data(),
BLOCK_SIZE);
                    if (bytes_read < 0) {
                        std::cerr << "Error reading from file: " <<
std::strerror(errno) << std::endl;
                        lab2_close(fd);
                        return;
                    }
                }

                lab2_close(fd);
            } else {
                // Standard file IO with O_DIRECT
                const int fd = open(file_path.c_str(), O_RDONLY |
O_DIRECT);
                if (fd < 0) {
                    std::cerr << "Error opening file for random read
benchmark!" << std::endl;
                    return;
                }

                void* aligned_buffer = nullptr;
                if (posix_memalign(&aligned_buffer, 512, BLOCK_SIZE) != 0)
{
                    std::cerr << "Error allocating aligned buffer!" <<
std::endl;
                    close(fd);
                    return;
                }

                if (lseek(fd, random_offset, SEEK_SET) < 0) {
                    std::cerr << "Error seeking in file: " <<
std::strerror(errno) << std::endl;
                    free(aligned_buffer);
                    close(fd);
                    return;
                }

                for (int j = 0; j < 2048; ++j) {
                    ssize_t bytes_read = read(fd, aligned_buffer,
BLOCK_SIZE);
                    if (bytes_read < 0) {
                        std::cerr << "Error reading from file: " <<
std::strerror(errno) << std::endl;
                        free(aligned_buffer);
                        close(fd);
                        return;
                    }
                }

                free(aligned_buffer);
                close(fd);
            }

            auto end = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double> duration = end - start;
            durations.push_back(duration.count());

            if (verbose) {
```

```
                std::cout << "Random Read Iteration " << i + 1
                          << ": Latency = " << duration.count() << "
seconds"
                          << std::endl;
            }
        }

        const double avg_duration = std::accumulate(durations.begin(),
durations.end(), 0.0) / durations.size();
        const double min_duration = *std::ranges::min_element(durations);
        const double max_duration = *std::ranges::max_element(durations);

        std::cout << "=== Random Read Benchmark Results ===\n";
        std::cout << "Average read latency: " << avg_duration << "
seconds\n";
        std::cout << "Minimum read latency: " << min_duration << "
seconds\n";
        std::cout << "Maximum read latency: " << max_duration << "
seconds\n";

        if (use_cache) {
            std::cout << "Cache hits: " << lab2_get_cache_hits() <<
std::endl;
            std::cout << "Cache misses: " << lab2_get_cache_misses() <<
std::endl;
            lab2_reset_cache_counters();
        }
    }

}
```

API блочного кэша

```
#include "lab2_cache.h"
#include <fcntl.h>
#include <unistd.h>
#include <map>
#include <list>
#include <vector>
#include <cstring>
#include <iostream>
#include <cerrno>

#define BLOCK_SIZE 4096     // Block size in bytes
#define MAX_CACHE_SIZE 32 // Max blocks in cache
// 4096 * 32 = 128KB

// Global counters for cache hits and misses
static unsigned long cache_hits = 0;
static unsigned long cache_misses = 0;

struct CacheBlock {
    char* data;           // Pointer to block data
    bool is_dirty;        // Dirty flag (modified since read)
    bool was_accessed;    // Reference bit for Second-Chance
};

struct FileDescriptor {
    int fd;
    off_t offset;
};

typedef std::pair<int, off_t> CacheKey; // (file descriptor, block id
```

```
corresponding to offset)

std::map<int, FileDescriptor> fd_table;     // Table for file descriptors
std::map<CacheKey, CacheBlock> cache_table;       // Table for cache blocks
std::list<CacheKey> cache_queue;            // Queue for Second-Chance

// Helper functions
FileDescriptor& get_file_descriptor(int fd);

/**
 * Frees a single cache block from memory.
 * Retrieves LChecks if a page has was_accessed bit, and if not,
 */
void free_cache_block() {
    while (!cache_queue.empty()) {
        CacheKey key = cache_queue.front();
        cache_queue.pop_front();
        CacheBlock& block = cache_table[key];
        if (block.was_accessed) {
            // Has been referenced, so reset reference bit and move to back
            block.was_accessed = false;
            cache_queue.push_back(key);
        } else {
            if (block.is_dirty) { // Save to disk if dirty before removing
                const int fd = key.first;
                const off_t block_id = key.second;
                ssize_t ret = pwrite(fd, block.data, BLOCK_SIZE, block_id *
BLOCK_SIZE);
                if (ret != BLOCK_SIZE) {
                    perror("pwrite");
                }
            }
            // Remove from cache
            free(block.data);
            cache_table.erase(key);
            break;
        }
    }
}

/**
 * Allocates an aligned buffer into memory.
 * @return Pointer to buffer
 */
char* allocate_aligned_buffer() {
    void* buf = nullptr;
    if (posix_memalign(&buf, BLOCK_SIZE, BLOCK_SIZE) != 0) {
        perror("posix_memalign");
        return nullptr;
    }
    return static_cast<char*>(buf);
}

/**
 * Opens a file with direct I/O.
 * @param path Path to file
 * @param flags File flags, same as in open()
 * @param mode File mode, same as in open()
 * @return Integer file descriptor
 */
int lab2_open(const char* path, const int flags, const mode_t mode) {
    const int fd = open(path, flags | O_DIRECT, mode);  // Read and write,
and bypass OS cache
    if (fd < 0) {
        perror("open");
```

```
        return -1;
    }
    // For simplicity, use actual fd
    fd_table[fd] = {fd, 0};
    return fd;
}

/**
 * Closes a file.
 * @param fd File descriptor
 * @return 0 on success, -1 on failure
 */
int lab2_close(const int fd) {
    const auto iterator = fd_table.find(fd);
    if (iterator == fd_table.end()) {
        errno = EBADF;                          // Bad file descriptor
        return -1;
    }
    lab2_fsync(fd);                             // Sync data before
closing
    const int result = close(iterator->second.fd);
    fd_table.erase(iterator);
    return result;
}

/**
 * Reads data from a file into a buffer.
 * @param fd File descriptor
 * @param buf Buffer to read into
 * @param count Number of bytes to read
 * @return Number of bytes read on success, -1 on failure
 */
ssize_t lab2_read(const int fd, void* buf, const size_t count) {
    auto& [found_fd, offset] = get_file_descriptor(fd);
    if (found_fd < 0 || offset < 0) {
        errno = EBADF;
        return -1;
    }
    size_t bytes_read = 0;
    const auto buffer = static_cast<char*>(buf);

    while (bytes_read < count) {
        off_t block_id = offset / BLOCK_SIZE;
        const size_t block_offset = offset % BLOCK_SIZE; // Offset within
block
        const size_t bytes_to_read = std::min(BLOCK_SIZE - block_offset,
count - bytes_read); // To read from block

        // Check if block is in cache
        CacheKey key = {found_fd, block_id};
        auto cache_iterator = cache_table.find(key);
        if (cache_iterator != cache_table.end()) {
            // HIT: Read from cache
            cache_hits++;

            CacheBlock& found_block = cache_iterator->second;
            found_block.was_accessed = true;
            size_t available_bytes = BLOCK_SIZE - block_offset; // What's
available in block
            const size_t bytes_from_block = std::min(bytes_to_read,
available_bytes); // Compare what we need with what's available
            std::memcpy(buffer + bytes_read, found_block.data +
block_offset, bytes_from_block);
            offset += bytes_from_block;
            bytes_read += bytes_from_block;
```

```
        } else {
            // MISS: Read block from disk
            cache_misses++;

            if (cache_table.size() >= MAX_CACHE_SIZE) { // Cache is full,
evict a block
                free_cache_block();
            }

            char* aligned_buf = allocate_aligned_buffer();
            const ssize_t ret = pread(found_fd, aligned_buf, BLOCK_SIZE,
block_id * BLOCK_SIZE);
            if (ret < 0) {
                // Failed to read
                perror("pread");
                free(aligned_buf);
                return -1;
            }
            if (ret == 0) {
                // EOF reached
                free(aligned_buf);
                break; // Exit the loop as there's no more data to read
            }
            // Partial read or full block read
            const auto valid_data_size = static_cast<size_t>(ret);
            // Fill the rest of the block with zeros if partial read
            if (ret < BLOCK_SIZE) {
                std::memset(aligned_buf + ret, 0, BLOCK_SIZE - ret);
            }
            // Add new block to cache
            const CacheBlock new_block = {aligned_buf, false, true};
            cache_table[key] = new_block;
            cache_queue.push_back(key);

            size_t available_bytes = valid_data_size - block_offset; //
What's available in block
            if (available_bytes <= 0) {
                // No valid data available after the offset
                break;
            }
            const size_t bytes_from_block = std::min(bytes_to_read,
available_bytes); // Compare what we need with what's available
            std::memcpy(buffer + bytes_read, aligned_buf + block_offset,
bytes_from_block);
            offset += bytes_from_block;
            bytes_read += bytes_from_block;
        }
    }
    return bytes_read;
}

/**
 * Writes data from a buffer to a file.
 * @param fd File descriptor
 * @param buf Buffer to write from
 * @param count Number of bytes to write
 * @return Number of bytes written on success, -1 on failure
 */
ssize_t lab2_write(const int fd, const void* buf, const size_t count) {
    auto& [found_fd, offset] = get_file_descriptor(fd);
    if (found_fd < 0 || offset < 0) {
        errno = EBADF;
        return -1;
    }
    size_t bytes_written = 0;
```

```
    const auto buffer = static_cast<const char*>(buf);

    while (bytes_written < count) {
        off_t block_id = offset / BLOCK_SIZE;
        const size_t block_offset = offset % BLOCK_SIZE;
        const size_t to_write = std::min(BLOCK_SIZE - block_offset, count -
bytes_written);

        // Check if block is in cache
        CacheKey key = {found_fd, block_id};
        auto cache_it = cache_table.find(key);
        CacheBlock* block_ptr = nullptr;

        if (cache_it == cache_table.end()) { // MISS
            cache_misses++;

            // If cache is full, evict a block
            if (cache_table.size() >= MAX_CACHE_SIZE) {
                free_cache_block();
            }
            // Read the block from disk to cache
            char* aligned_buf = allocate_aligned_buffer();
            ssize_t ret = pread(found_fd, aligned_buf, BLOCK_SIZE, block_id
* BLOCK_SIZE);
            if (ret < 0) {
                // Read failed
                perror("pread");
                free(aligned_buf);
                return -1;
            } else if (ret == 0) {
                // If read fails (e.g., beyond EOF), fill with zeros
                std::memset(aligned_buf, 0, BLOCK_SIZE);
            } else if (ret < BLOCK_SIZE) {
                // Partial read, fill rest with zeros
                std::memset(aligned_buf + ret, 0, BLOCK_SIZE - ret);
            }
            // Add read block to cache
            CacheBlock& block = cache_table[key] = {aligned_buf, false,
true};
            cache_queue.push_back(key);
            block_ptr = &block;
        } else { // HIT
            cache_hits++;
            block_ptr = &cache_it->second;
            block_ptr->was_accessed = true;
        }
        // Write to cache
        std::memcpy(block_ptr->data + block_offset, buffer + bytes_written,
to_write);
        block_ptr->is_dirty = true;

        offset += to_write;
        bytes_written += to_write;
    }
    return bytes_written;
}

/**
 * Sets file's offset. Only supports SEEK_SET (absolute positioning).
 * @param fd File descriptor
 * @param offset Absolute offset in bytes
 * @param whence Positioning mode (only SEEK_SET supported)
 * @return New file's offset on success, -1 on failure
 */
off_t lab2_lseek(const int fd, const off_t offset, const int whence) {
```

```
    auto& [found_fd, file_offset] = get_file_descriptor(fd);
    if (found_fd < 0 || file_offset < 0) {
        errno = EBADF;
        return -1;
    }
    if (whence != SEEK_SET) {
        errno = EINVAL;
        return -1;
    }
    if (offset < 0) {
        errno = EINVAL;
        return -1;
    }
    file_offset = offset;
    return file_offset;
}

/**
 * Flushes dirty blocks to disk for a file descriptor.
 * @param fd_to_sync File descriptor to sync
 * @return 0 on success, -1 on failure
 */
int lab2_fsync(const int fd_to_sync) {
    const int found_fd = get_file_descriptor(fd_to_sync).fd;
    if (found_fd < 0) {
        errno = EBADF;
        return -1;
    }
    // Write back dirty blocks for this file descriptor
    for (auto& [key, block] : cache_table) {
        if (key.first == found_fd && block.is_dirty) {
            ssize_t ret = pwrite(found_fd, block.data, BLOCK_SIZE,
key.second * BLOCK_SIZE);
            if (ret != BLOCK_SIZE) {
                perror("pwrite");
                return -1;
            }
            block.is_dirty = false;
        }
    }
    // Ensure all data and metadata are flushed to the physical storage
    return fsync(found_fd);
}

/**
 * Gets the file descriptor struct for a given file descriptor id.
 * @param fd File descriptor
 * @return FileDescriptor struct
 */
FileDescriptor& get_file_descriptor(const int fd) {
    const auto iterator = fd_table.find(fd);
    if (iterator == fd_table.end()) {
        static FileDescriptor invalid_fd = {-1, -1};
        return invalid_fd;
    }
    return iterator->second;
}

unsigned long lab2_get_cache_hits() {
    return cache_hits;
}

unsigned long lab2_get_cache_misses() {
    return cache_misses;
}
```

```
void lab2_reset_cache_counters() {
    cache_hits = 0;
    cache_misses = 0;
}
```

Результаты:

Write без кэша (Последовательная запись)

```
Overall Statistics:
Average write latency: 0.0822461 seconds
Minimum write latency: 0.0786764 seconds
Maximum write latency: 0.0933617 seconds
```

% time	seconds	usecs/call	calls	errors	syscall
99.58	1.246773	6	204805		write
0.19	0.002435	24	100		unlink
0.10	0.001298	12	100		fsync
0.10	0.001225	11	111	5	openat
0.03	0.000349	3	106		close
0.00	0.000000	0	5		read
0.00	0.000000	0	7		fstat
0.00	0.000000	0	26		mmap
0.00	0.000000	0	7		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	2		pread64
0.00	0.000000	0	1	1	access
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		futex
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	2	2	newfstatat
0.00	0.000000	0	1		set_robust_list
0.00	0.000000	0	1		prlimit64
0.00	0.000000	0	1		getrandom
0.00	0.000000	0	1		rseq
100.00	1.252080	6	205284	8	total

Write с кэшем (Последовательная запись)

```
Overall Statistics:
Average write latency: 0.00752635 seconds
Minimum write latency: 0.00663992 seconds
Maximum write latency: 0.00988521 seconds
Cache hits: 179200
Cache misses: 25600
```

% time	seconds	usecs/call	calls	errors	syscall
79.06	0.126326	4	25600		pwrite64
18.59	0.029700	1	25602		pread64
0.94	0.001499	14	100		unlink
0.87	0.001393	6	200		fsync
0.42	0.000679	6	111	5	openat
0.11	0.000175	1	106		close
0.01	0.000010	1	7		write
0.00	0.000002	0	5		brk
0.00	0.000000	0	5		read
0.00	0.000000	0	7		fstat
0.00	0.000000	0	26		mmap
0.00	0.000000	0	7		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	1	1	access
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		futex
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	2	2	newfstatat
0.00	0.000000	0	1		set_robust_list
0.00	0.000000	0	1		prlimit64
0.00	0.000000	0	1		getrandom
0.00	0.000000	0	1		rseq
100.00	0.159784	3	51788	8	total

Read без кэша (Последовательное чтение)

```
Overall Statistics:
Average read latency: 0.0300317 seconds
Minimum read latency: 0.0288578 seconds
Maximum read latency: 0.0344697 seconds
```

% time	seconds	usecs/call	calls	errors	syscall
99.90	0.759692	3	204905		read
0.07	0.000514	4	111	5	openat
0.03	0.000246	2	106		close
0.00	0.000010	2	5		write
0.00	0.000000	0	7		fstat
0.00	0.000000	0	26		mmap
0.00	0.000000	0	7		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	2		pread64
0.00	0.000000	0	1	1	access
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		futex
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	2	2	newfstatat
0.00	0.000000	0	1		set_robust_list
0.00	0.000000	0	1		prlimit64
0.00	0.000000	0	1		getrandom
0.00	0.000000	0	1		rseq
100.00	0.760462	3	205184	8	total

Read с кэшем (Последовательное чтение)

```
Overall Statistics:
Average read latency: 0.009879 seconds
Minimum read latency: 0.00938243 seconds
Maximum read latency: 0.0126887 seconds
Cache hits: 179200
Cache misses: 25700
```

% time	seconds	usecs/call	calls	errors	syscall
98.65	0.059175	2	25702		pread64
0.40	0.000241	2	111	5	openat
0.34	0.000201	2	100		fsync
0.28	0.000169	6	26		mmap
0.20	0.000118	1	106		close
0.04	0.000021	3	7		fstat
0.03	0.000015	3	5		read
0.02	0.000014	2	5		brk
0.02	0.000011	1	7		mprotect
0.01	0.000005	5	1		munmap
0.01	0.000005	2	2	2	newfstatat
0.01	0.000004	0	7		write
0.01	0.000004	4	1	1	access
0.00	0.000002	2	1		futex
0.00	0.000001	1	1		prlimit64
0.00	0.000001	1	1		getrandom
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		set_robust_list
0.00	0.000000	0	1		rseq
100.00	0.059987	2	26088	8	total

Write без кэша (Случайная запись)

```
Average write latency: 0.0306145 seconds
Minimum write latency: 0.0288675 seconds
Maximum write latency: 0.0560978 seconds
```

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.70    0.902319           4    204805           write
  0.15    0.001375          13       100           fsync
  0.06    0.000558           5       111         5 openat
  0.04    0.000336           3       106           close
  0.02    0.000164           6        26           mmap
  0.02    0.000163           1       100           lseek
  0.00    0.000043           6         7           mprotect
  0.00    0.000025           3         7           fstat
  0.00    0.000014           2         5           read
  0.00    0.000014          14         1           munmap
  0.00    0.000008           2         3         2 newfstatat
  0.00    0.000006           2         3           brk
  0.00    0.000004           2         2           pread64
  0.00    0.000003           3         1           arch_prctl
  0.00    0.000003           3         1           set_robust_list
  0.00    0.000003           3         1           getrandom
  0.00    0.000002           2         1           futex
  0.00    0.000002           2         1           set_tid_address
  0.00    0.000002           2         1           prlimit64
  0.00    0.000002           2         1           rseq
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.905046           4    205285         8 total
```

Write с кэшем (Случайная запись)

```
Average write latency: 0.0192045 seconds
Minimum write latency: 0.0130736 seconds
Maximum write latency: 0.0303452 seconds
Cache hits: 179140
Cache misses: 25660
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 51.35    0.083012           3     25689           pwrite64
 47.27    0.076421           2     25662           pread64
  0.88    0.001421           7       200           fsync
  0.24    0.000396           3       111         5 openat
  0.12    0.000202           1       106           close
  0.07    0.000107           4        26           mmap
  0.02    0.000027           3         7           mprotect
  0.01    0.000020           2         7           write
  0.01    0.000011           2         5           read
  0.01    0.000011           1         7           fstat
  0.00    0.000008           1         5           brk
  0.00    0.000005           5         1           munmap
  0.00    0.000002           2         1           arch_prctl
  0.00    0.000002           2         1           set_tid_address
  0.00    0.000002           0         3         2 newfstatat
  0.00    0.000002           2         1           set_robust_list
  0.00    0.000002           2         1           getrandom
  0.00    0.000002           2         1           rseq
  0.00    0.000001           1         1           futex
  0.00    0.000001           1         1           prlimit64
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.161655           3     51838         8 total
```

Read без кэша (Случайное чтение)

```
Average read latency: 0.0232364 seconds
Minimum read latency: 0.00130098 seconds
Maximum read latency: 0.0345497 seconds
```

% time	seconds	usecs/call	calls	errors	syscall
99.84	0.708648	3	204805		read
0.05	0.000357	3	111	5	openat
0.03	0.000187	1	106		close
0.02	0.000172	6	26		mmap
0.02	0.000165	165	1		execve
0.02	0.000127	1	100		lseek
0.01	0.000039	5	7		mprotect
0.00	0.000019	2	7		fstat
0.00	0.000014	14	1		munmap
0.00	0.000011	2	5		write
0.00	0.000009	3	3		brk
0.00	0.000005	2	2		pread64
0.00	0.000005	1	3	2	newfstatat
0.00	0.000004	4	1	1	access
0.00	0.000003	3	1		arch_prctl
0.00	0.000003	3	1		prlimit64
0.00	0.000003	3	1		getrandom
0.00	0.000002	2	1		futex
0.00	0.000002	2	1		set_tid_address
0.00	0.000002	2	1		set_robust_list
0.00	0.000002	2	1		rseq
100.00	0.709779	3	205185	8	total

Read с кэшем (Случайное чтение)

```
Average read latency: 0.00947416 seconds
Minimum read latency: 0.00708904 seconds
Maximum read latency: 0.0184305 seconds
Cache hits: 163794
Cache misses: 41006
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 98.50    0.075136           1     41008           pread64
  0.40    0.000303           3       100           fsync
  0.38    0.000289           2       111         5 openat
  0.27    0.000204         204         1           execve
  0.19    0.000146           1       106           close
  0.18    0.000137           5        26           mmap
  0.02    0.000017           2         7           fstat
  0.02    0.000015           3         5           read
  0.01    0.000011           2         5           brk
  0.01    0.000008           2         3         2 newfstatat
  0.01    0.000005           5         1         1 access
  0.01    0.000004           0         7           write
  0.00    0.000002           2         1           getrandom
  0.00    0.000000           0         7           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         1           futex
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           prlimit64
  0.00    0.000000           0         1           rseq
------ ----------- ----------- --------- --------- ----------------
100.00    0.076277           1     41395         8 total
```

Сравнение производительности

Запущены программы-загрузчики для чтения и записи данных в файл размером 1 МБ (1,048,576 байт) с использованием блоков размером 512 байт. Тесты проводились как с использованием блочного кэша, так и без него, для оценки влияния кэширования на производительность и количество системных вызовов.

Размеры блоков и кэша

- Размер блока в бенчмарках: 512 байт
- Размер блока в кэше: 4096 байт
- Количество блоков в файле:
  - При размере блока 512 байт: 1,048,576 / 512 = 2048 блоков
  - При размере блока 4096 байт: 1,048,576 / 4096 = 256 блоков
- Максимальный размер кэша: 1024 блока (1024 * 4096 байт = 4,194,304 байт или 4 МБ)

Запись без кэша

- Среднее время записи: 0.0822461 секунд
- Количество системных вызовов write: 204 805

Ожидалось, что количество записей будет 1 048 576 / 512 = 2048 вызовов write. При 100 итерациях: 204 800, что близко к реальному значению.

Запись с кэшем

- Среднее время записи: 0.0075996 секунд
- Количество системных вызовов pwrite64: 25 600
- Количество системных вызовов pread64: 25 602

Ожидалось, что количество записей будет 1 048 576 / 4096 = 256 вызовов pwrite64. При 100 итерациях: 25 600. Время записи с кэшем уменьшилось в 10 раз, что, логично учитывая меньшее количество обращений к диску.

Чтение без кэша

- Среднее время чтения: 0.0300317 секунд
- Количество системных вызовов read: 204 905

Аналогично ожидается 2048 вызовов read * 100 итераций = 204 800.

Чтение с кэшем

- Среднее время чтения: 0.00879 секунд
- Количество системных вызовов pread64: 25 702

Ожидаемое количество вызовов pread64 = 256 * 100 = 25 600. Время чтения с кэшем уменьшилось в 3.5 раза.

Заключение

В ходе выполнения проекта был разработан блочный кэш в пространстве пользователя с политикой вытеснения Second-Chance. Проведенные тесты показали значительное улучшение производительности при использовании кэша как для операций чтения, так и для операций записи.