Университет ИТМО

Факультет Программной Инженерии и Компьютерной техники

Операционные системы

Лабораторная работа №2

Вариант Windows, LRU

Выполнил:

Кукольников Е. С., P3315

Преподаватель:

Романов А. И.

Санкт-Петербург
2025

Оглавление

Задание

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.

Краткий обзор кода

Используемые структуры

```
// Размер блока
#define BLOCK_SIZE 4096
// Макс кол-во блоков
#define MAX_BLOCKS_IN_CACHE 180

// Статистика работы кэша
struct CacheStats {
    size_t cache_hits;
    size_t cache_misses;
};

// Инициализация статистики
CacheStats cache_stats {0, 0};

// Кэшблок
struct CacheBlock {
    char* data;          // Указатель на данные
    bool dirty_data;     // Флаг "грязных" данных
    ptrdiff_t useful_data; // Количество полезных данных в блоке
    ULONGLONG last_used;   // Время последнего использования (для LRU)
};

// Файловый дескриптор для Windows
struct FileDescriptor {
    HANDLE fd; // В Windows используется HANDLE
    int offset; // Смещение в файле
};
```

```
// Пара - HANDLE / id блока, соответствующий отступу в файле
typedef std::pair<HANDLE, LARGE_INTEGER> CacheKey;

// Таблица блоков кэша
std::map<CacheKey, CacheBlock, CacheKeyComparator> cache_table;

// Таблица файловых дескрипторов
std::map<HANDLE, FileDescriptor> fd_table;
```

Вспомогательные функции

```
// Выделение памяти под кэшблок (адреса выравнены)
char* allocate_aligned_buffer() {
    void* buf = _aligned_malloc(BLOCK_SIZE, BLOCK_SIZE);
    if (!buf) {
        DWORD error = GetLastError();
        std::cerr << "Cant allocate aligned buffer. Windows error code:
" << error << std::endl;
        return nullptr;
    }
    return static_cast<char*>(buf);
}

// Запись кэшблока на диск
int write_cache_block(HANDLE fd, void* buf, int count, int start_pos) {
    DWORD bytesWritten;
    OVERLAPPED overlapped = {0};
    overlapped.Offset = start_pos;

    if (!WriteFile(fd, buf, count, &bytesWritten, &overlapped)) {
        std::cerr << "Error writing the block cache: " << GetLastError()
<< std::endl;
        return -1;
    }

    if (bytesWritten != static_cast<DWORD>(count)) {
        std::cerr << "Error: Less data was recorded than expected\n";
        return -1;
    }

    return 0;
}

// Освобождение кэшблока
void free_cache_block(HANDLE found_fd) {
    if (cache_table.empty()) {
        return; // Если кэш пуст, ничего не делаем
    }

    // Находим блок, который использовался давно (LRU)
    auto lru_block = cache_table.begin();
    for (auto it = cache_table.begin(); it != cache_table.end(); ++it) {
        if (it->first.first == found_fd && it->second.last_used <
lru_block->second.last_used) {
```

```
            lru_block = it;
        }
    }

    // Если найденный блок принадлежит переданному файловому дескриптору
    if (lru_block->first.first == found_fd) {
        // Если данные "грязные", записываем их на диск
        if (lru_block->second.dirty_data) {
            if (write_cache_block(
                    found_fd,
                    lru_block->second.data,
                    static_cast<int>(lru_block->second.useful_data),
                    lru_block->first.second.QuadPart) != 0) {
                std::cerr << "Ошибка: не удалось записать блок на диск
(free_cache_block)\n";
                return;
                    }
            lru_block->second.dirty_data = false; // Сбрасываем флаг
"грязных" данных
        }

        // Освобождаем память, выделенную для данных
        if (lru_block->second.data != nullptr) {
            _aligned_free(lru_block->second.data);
            lru_block->second.data = nullptr;
        }

        // Удаляем блок из кэш-таблицы
        cache_table.erase(lru_block);
    }
}

// Освобождение всех кэшблоков
void free_all_cache_blocks() {
    if (cache_table.empty()) {
        return; // Если кэш пуст, ничего не делаем
    }

    // Проходим по всем элементам кэш-таблицы
    auto it = cache_table.begin();
    while (it != cache_table.end()) {
        if (it->second.data != nullptr) {
            _aligned_free(it->second.data);
            it->second.data = nullptr;
        }
        // Удаляем блок из кэш-таблицы
        it = cache_table.erase(it); // erase возвращает итератор на
следующий элемент
    }
}

// Получаем файловый дескриптор
FileDescriptor& get_file_descriptor(const HANDLE fd) {
    const auto iterator = fd_table.find(fd);
```

```
    if (iterator == fd_table.end()) {
        static FileDescriptor invalid_fd = {nullptr, {-1 } };
        return invalid_fd;
    }
    return iterator->second;
}
```

Реализация API для работы с файлами

```
// Открытие файла
HANDLE lab2_open(const char* path) {
    // Открываем файл с помощью CreateFile
    HANDLE fd = CreateFile(
        path,                                // Имя файла
        GENERIC_READ | GENERIC_WRITE,        // Доступ на чтение и запись
        0,                                   // Не разделяем доступ
        NULL,                                // Без атрибутов безопасности
        OPEN_EXISTING,                       // Открываем существующий файл
        FILE_ATTRIBUTE_NORMAL,               // Обычные атрибуты файла
        NULL                                 // Без шаблона файла
    );

    if (fd == INVALID_HANDLE_VALUE) {
        std::cerr << "Can't open file: " << path << "\n";
        return INVALID_HANDLE_VALUE;
    }

    // Инициализируем структуру FileDescriptor
    FileDescriptor fileDesc;
    fileDesc.fd = fd;
    fileDesc.offset = 0; // Начальное смещение в файле

    // Сохраняем информацию о файле в fd_table
    fd_table[fd] = fileDesc;
    // Возвращаем HANDLE
    return fd;
}

// Закрытие файла
int lab2_close(const HANDLE fd) {
    // Проверяем, есть ли дескриптор в таблице
    auto it = fd_table.find(fd);
    if (it == fd_table.end()) {
        std::cerr << "Invalid file descriptor\n";
        return -1;
    }

    // Синхронизируем данные перед закрытием
    lab2_fsync(fd);

    // Закрываем файл
    if (!CloseHandle(fd)) {
        std::cerr << "Failed to close file\n";
        return -1;
```

```
    }

    // Удаляем запись из fd_table
    fd_table.erase(it);
    return 0; // Успешное закрытие
}

// Чтение из файла
ptrdiff_t lab2_read(const HANDLE fd, void *buf, const size_t count) {
    // Получаем файловый дескриптор и смещение
    FileDescriptor& file_desc = get_file_descriptor(fd);
    if (file_desc.fd == INVALID_HANDLE_VALUE || file_desc.offset < 0 ||
!buf) {
        SetLastError(ERROR_INVALID_PARAMETER); // Устанавливаем ошибку
"Invalid parameter"
        return -1;
    }

    ptrdiff_t bytes_read = 0;
    const auto buffer = static_cast<char*>(buf);

    while (bytes_read < count) {
        // Получаем id блока, в который будем читать
        LARGE_INTEGER block_id;
        block_id.QuadPart = file_desc.offset / BLOCK_SIZE;

        // Отступ внутри кэшблока
        const size_t block_offset = file_desc.offset % BLOCK_SIZE;

        // Сколько байт прочтём на данной итерации
        const int iteration_read = static_cast<int>(std::min<ptrdiff_t>(
            static_cast<ptrdiff_t>(BLOCK_SIZE - block_offset),
            static_cast<ptrdiff_t>(static_cast<ptrdiff_t>(count) -
bytes_read)
        ));
        // Смотрим, есть ли блок в кэше
        CacheKey key = {fd, block_id};
        auto cache_iterator = cache_table.find(key);
        size_t bytes_from_block;

        if (cache_iterator != cache_table.end()) {
            // Попали в кэшблоки
            cache_stats.cache_hits++;

            CacheBlock& found_block = cache_iterator->second;

            found_block.last_used = GetTickCount64();
            // Получаем количество байт, которое можем прочесть
            ptrdiff_t available_bytes = found_block.useful_data -
static_cast<ptrdiff_t>(block_offset);

            // Если ничего прочесть не можем
            if (available_bytes <= 0) {
                break;
            }
```

```
            // Берём минимум из: сколько байт можем прочесть / сколько
надо
            bytes_from_block =
static_cast<size_t>(std::min<size_t>(iteration_read,
static_cast<int>(available_bytes)));

            // Копируем данные из кэша
            memcpy(buffer + bytes_read, found_block.data + block_offset,
bytes_from_block);
        } else {
            // Не попали в кэшблоки
            cache_stats.cache_misses++;

            // Если место закончилось, то удаляем какой-нибудь уже
существующий кэшблок
            if (cache_table.size() >= MAX_BLOCKS_IN_CACHE) {
                free_cache_block(fd);
            }

            // Создаём будущий кэшблок и читаем в него
            char* aligned_buf = allocate_aligned_buffer();
            if (!aligned_buf) {
                return -1; // Ошибка выделения памяти
            }

            // Читаем данные из файла
            LARGE_INTEGER read_offset;
            read_offset.QuadPart = block_id.QuadPart * BLOCK_SIZE;

            DWORD bytesRead;
            if (!ReadFile(fd, aligned_buf, BLOCK_SIZE, &bytesRead,
nullptr) || bytesRead == 0) {
                free(aligned_buf);
                break; // Ошибка чтения или конец файла
            }

            // Создаём блок, записываем в него, сколько данных мы прочли
            CacheBlock new_block = {aligned_buf, false,
static_cast<ptrdiff_t>(bytesRead), GetTickCount64()};
            cache_table[key] = new_block;

            // Смотрим, сколько байт сможем прочесть
            int available_bytes = static_cast<int>(bytesRead) -
static_cast<int>(block_offset);
            if (available_bytes <= 0) {
                break;
            }

            // Записываем данные из только что созданного кэшблока
            bytes_from_block =
static_cast<size_t>(std::min<size_t>(available_bytes, iteration_read));
            memcpy(buffer + bytes_read, aligned_buf + block_offset,
bytes_from_block);
        }
```

```
        // Фиксируем результаты итерации
        file_desc.offset += bytes_from_block;
        bytes_read += bytes_from_block;
    }

    return bytes_read;
}

//Запись в файл
ptrdiff_t lab2_write(const HANDLE fd, const void* buf, const size_t
count) {
    // Получаем файловый дескриптор и смещение
    FileDescriptor& file_desc = get_file_descriptor(fd);
    if (file_desc.fd == INVALID_HANDLE_VALUE || file_desc.offset < 0 ||
!buf) {
        SetLastError(ERROR_INVALID_PARAMETER); // Устанавливаем ошибку
"Invalid parameter"
        return -1;
    }

    ptrdiff_t bytes_written = 0;
    const auto buffer = static_cast<const char*>(buf);

    while (bytes_written < count) {
        // Получаем id блока, в который будем писать
        LARGE_INTEGER block_id;
        block_id.QuadPart = file_desc.offset / BLOCK_SIZE;

        // Отступ внутри кэшблока
        const size_t block_offset = file_desc.offset % BLOCK_SIZE;

        // Сколько байт запишем на данной итерации
        const int iteration_write =
static_cast<int>(std::min<ptrdiff_t>(
                    static_cast<ptrdiff_t>(BLOCK_SIZE - block_offset),
                    static_cast<ptrdiff_t>(static_cast<ptrdiff_t>(count)
- bytes_written)
                ));
        // Смотрим, есть ли блок в кэше
        CacheKey key = {fd, block_id};
        auto cache_iterator = cache_table.find(key);
        CacheBlock* block_ptr;

        if (cache_iterator == cache_table.end()) {
            // Не попали в кэшблоки
            cache_stats.cache_misses++;

            // Освобождаем место, если закончилось
            if (cache_table.size() >= MAX_BLOCKS_IN_CACHE) {
                free_cache_block(fd);
            }

            // Создаём кэшблок, записываем в него данные из файла
            char* aligned_buf = allocate_aligned_buffer();
```

```
            if (!aligned_buf) {
                return -1; // Ошибка выделения памяти
            }

            // Читаем данные из файла
            LARGE_INTEGER read_offset;
            read_offset.QuadPart = block_id.QuadPart * BLOCK_SIZE;

            DWORD bytesRead;
            if (!ReadFile(fd, aligned_buf, BLOCK_SIZE, &bytesRead,
nullptr) || bytesRead == 0) {
                free(aligned_buf);
                break; // Ошибка чтения или конец файла
            }

            // Создаём блок, записываем в него, сколько данных мы прочли
            CacheBlock new_block = {aligned_buf, false,
static_cast<ptrdiff_t>(bytesRead), GetTickCount64()};
            cache_table[key] = new_block;
            block_ptr = &cache_table[key];
        } else {
            // Попали в кэшблоки
            cache_stats.cache_hits++;
            block_ptr = &cache_iterator->second;
        }

        // Записываем в кэшблок, теперь он содержит грязные данные
        memcpy(block_ptr->data + block_offset, buffer + bytes_written,
iteration_write);
        block_ptr->dirty_data = true;
        // Обновляем время последнего использования
        block_ptr->last_used = GetTickCount64();
        // Обновляем useful_data - мы могли записать чуть больше, чем
было записано в блок раньше
        block_ptr->useful_data =
static_cast<ptrdiff_t>(std::max<ptrdiff_t>(
            block_ptr->useful_data,
            static_cast<ptrdiff_t>(block_offset + iteration_write)));

        // Фиксируем результаты итерации
        file_desc.offset += iteration_write;
        bytes_written += iteration_write;
    }

    return bytes_written;
}

// Перестановка позиции указателя
int lab2_lseek(const HANDLE fd, const int offset, const int whence) {
    auto& [found_fd, file_offset] = get_file_descriptor(fd);

    if (found_fd == INVALID_HANDLE_VALUE || file_offset < 0) {
        SetLastError(ERROR_INVALID_HANDLE);
        int error_offset;
        error_offset = -1;
```

```
        return error_offset;
    }

    if (whence != SEEK_SET || offset < 0) {
        SetLastError(ERROR_INVALID_PARAMETER);
        int error_offset;
        error_offset = -1;
        return error_offset;
    }

    file_offset = offset;
    return file_offset;
}

// Синхронизация данных
int lab2_fsync(HANDLE fd) {
    // Получаем файловый дескриптор
    FileDescriptor& file_desc = get_file_descriptor(fd);
    if (file_desc.fd == INVALID_HANDLE_VALUE) {
        SetLastError(ERROR_INVALID_HANDLE); // Устанавливаем ошибку
"Invalid handle"
        return -1;
    }

    // Проходим по всем блокам в кэше
    for (auto& [key, block] : cache_table) {
        // Если блок принадлежит текущему файловому дескриптору и
помечен как "грязный"
        if (key.first == fd && block.dirty_data) {
            block.last_used = GetTickCount64();
            // Записываем блок на диск
            if (write_cache_block(fd, block.data,
static_cast<int>(block.useful_data), key.second.QuadPart) != 0) {
                std::cerr << "Can't flush block (fsync)\n";
                return -1;
            }
            // Сбрасываем флаг "грязных" данных
            block.dirty_data = false;
        }
    }

    return 0;
}
```

Данные о работе до и после внедрения своего page cache

Чтение одного и того же блока данных:

```
Test #1 - Simple reading of the same large block 100000 times

Execution time without cache: 0.0438232 seconds.

Execution time with cache: 0.102265 seconds.
Cache hits: 899991, Cache miss: 9
```

```
Test #1 - Simple reading of the same large block 100000 times

Execution time without cache: 0.0258085 seconds.

Execution time with cache: 0.114223 seconds.
Cache hits: 899991, Cache miss: 9
```

```
Test #1 - Simple reading of the same large block 100000 times

Execution time without cache: 0.0269359 seconds.

Execution time with cache: 0.109538 seconds.
Cache hits: 899991, Cache miss: 9
```

Чтение случайного блока данных:

```
Test #2 - Reading a random block of data 100001 times

Execution time without cache: 0.0838068 seconds.

Execution time with cache: 0.0560185 seconds.
Cache hits: 319621, Cache miss: 76
```

```
Test #2 - Reading a random block of data 100001 times

Execution time without cache: 0.0286857 seconds.

Execution time with cache: 0.0571916 seconds.
Cache hits: 319456, Cache miss: 76
```

```
Test #2 - Reading a random block of data 100001 times

Execution time without cache: 0.0281979 seconds.

Execution time with cache: 0.0568587 seconds.
Cache hits: 319677, Cache miss: 76
```

Чтение из случайного места и запись результата в случайное место:

```
Test #3 - Reading from an arbitrary location and writing the result to an arbitrary location in the file 100001 times

Execution time without cache: 0.0549631 seconds.

Execution time with cache: 0.0388579 seconds.
Cache hits: 224087, Cache miss: 74
```

```
Test #3 - Reading from an arbitrary location and writing the result to an arbitrary location in the file 100001 times

Execution time without cache: 0.0591733 seconds.

Execution time with cache: 0.0456472 seconds.
Cache hits: 224250, Cache miss: 74
```

```
Test #3 - Reading from an arbitrary location and writing the result to an arbitrary location in the file 100001 times

Execution time without cache: 0.0605368 seconds.

Execution time with cache: 0.0451669 seconds.
Cache hits: 224163, Cache miss: 74
```

Работа с несколькими файлами одновременно:

```
Test #4 - Working with multiple files at the same time

Cache hits: 1, Cache miss: 3
```

Вывод

В результате выполнения лабораторной работы я познакомился с блочным кэшем в пространстве пользователя, а также с политикой вытеснения Least Recently Used. Анализируя результаты работы программы, можно сказать, что в случаях записи блоков данных, а также чтения случайных данных удалось получить выигрыш в скорости, но в простом чтении данных выигрыша в скорости достичь не получилось. Скорее всего это произошло из-за постоянного сравнения кэшблоков с целью определения наиболее давно используемого. Также скорость уменьшается из-за добавления новых кэшблоков в таблицу, сортируя значения типа HENDLE, которые являются уникальными идентификаторами файловых дескрипторов.