Федеральное государственное автономное образовательное учреждение высшего образования  
 «Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Дисциплина «Операционные системы»

**Лабораторная работа №2**Вариант ”FIFO”

*Выполнили*:

Студент группы P3306

Константинова Мария

*Преподаватель:*

Тюрин И.Н.

г. Санкт-Петербург

2024 год

## 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).

## Краткий обзор кода

**Вспомогательная функция: findCacheEntry**

CacheEntry\* findCacheEntry(FileDescriptor \*fd, int fileOffset) {

CacheEntry \*current = fd->cacheHead;

while (current) {

if (current->fileOffset == fileOffset) {

return current;

}

current = current->next;

}

return NULL;

}

* Функция ищет в кэше элемент с заданным смещением (fileOffset). Проходит по связанному списку, начиная с cacheHead.

**Функция lab2\_open**

DLL\_EXPORT int lab2\_open(const char \*path) {

HANDLE hFile = CreateFileA(path, GENERIC\_READ | GENERIC\_WRITE, 0, NULL,

OPEN\_ALWAYS, FILE\_FLAG\_NO\_BUFFERING | FILE\_FLAG\_WRITE\_THROUGH, NULL);

if (hFile == INVALID\_HANDLE\_VALUE) {

return -1;

}

for (int i = 0; i < 256; i++) {

if (openFiles[i] == NULL) {

openFiles[i] = (FileDescriptor\*)malloc(sizeof(FileDescriptor));

openFiles[i]->hFile = hFile;

openFiles[i]->cacheHead = NULL;

openFiles[i]->cacheTail = NULL;

openFiles[i]->cacheCount = 0;

return i;

}

}

CloseHandle(hFile);

return -1;

}

* **CreateFileA** открывает (или создаёт, если не существует) файл по указанному пути с флагами FILE\_FLAG\_NO\_BUFFERING | FILE\_FLAG\_WRITE\_THROUGH, что отключает кэш ОС.
* Если файл открыт успешно, ищется свободный слот в глобальном массиве openFiles.
* Если найден, выделяется память для структуры FileDescriptor, и инициализируются поля (дескриптор, пустой кэш).
* Возвращается индекс (fd) для использования в последующих вызовах.

**Функция lab2\_close**

DLL\_EXPORT int lab2\_close(int fd) {

if (fd < 0 || fd >= 256 || openFiles[fd] == NULL)

return -1;

FileDescriptor \*file = openFiles[fd];

CacheEntry \*current = file->cacheHead;

while (current) {

if (current->dirty) {

DWORD bytesWritten;

LARGE\_INTEGER li;

li.QuadPart = current->fileOffset;

SetFilePointerEx(file->hFile, li, NULL, FILE\_BEGIN);

WriteFile(file->hFile, current->data, BLOCK\_SIZE, &bytesWritten, NULL);

}

CacheEntry \*next = current->next;

VirtualFree(current->data, 0, MEM\_RELEASE);

free(current);

current = next;

}

CloseHandle(file->hFile);

free(file);

openFiles[fd] = NULL;

return 0;

}

* Проверяется корректность дескриптора.
* Для каждого элемента кэша:
  + Если данные изменены (dirty == 1), данные записываются на диск.
  + Затем освобождается выделенная память (с помощью VirtualFree) и память под элемент кэша.
* Закрывается файловый дескриптор и освобождается память для структуры FileDescriptor.
* Указатель в глобальном массиве обнуляется.

**Функция lab2\_read**

DLL\_EXPORT SSIZE\_T lab2\_read(int fd, void \*buf, size\_t count) {

if (fd < 0 || fd >= 256 || openFiles[fd] == NULL)

return -1;

FileDescriptor \*file = openFiles[fd];

int fileOffset = lab2\_lseek(fd, 0, SEEK\_CUR);

CacheEntry \*entry = findCacheEntry(file, fileOffset);

if (entry) {

memcpy(buf, entry->data, count);

return count;

}

// выровненный буфер через VirtualAlloc (под NO\_BUFFERING требования Windows)

char \*alignedBuffer = (char\*)VirtualAlloc(NULL, BLOCK\_SIZE, MEM\_COMMIT | MEM\_RESERVE, PAGE\_READWRITE);

if (!alignedBuffer)

return -1;

DWORD bytesRead;

LARGE\_INTEGER li;

li.QuadPart = fileOffset;

SetFilePointerEx(file->hFile, li, NULL, FILE\_BEGIN);

if (!ReadFile(file->hFile, alignedBuffer, BLOCK\_SIZE, &bytesRead, NULL)) {

VirtualFree(alignedBuffer, 0, MEM\_RELEASE);

return -1;

}

// новый элемент кэша в конец списка

CacheEntry \*newEntry = (CacheEntry\*)malloc(sizeof(CacheEntry));

newEntry->fileOffset = fileOffset;

newEntry->data = alignedBuffer;

newEntry->dirty = 0;

newEntry->next = NULL;

if (file->cacheTail)

file->cacheTail->next = newEntry;

file->cacheTail = newEntry;

if (!file->cacheHead)

file->cacheHead = newEntry;

file->cacheCount++;

if (file->cacheCount > CACHE\_SIZE) {

CacheEntry \*old = file->cacheHead;

file->cacheHead = old->next;

VirtualFree(old->data, 0, MEM\_RELEASE);

free(old);

file->cacheCount--;

}

memcpy(buf, alignedBuffer, count);

return bytesRead;

}

* Проверяется корректность дескриптора.
* Получается текущая позиция в файле (с помощью lab2\_lseek).
* Если в кэше уже есть блок с данным смещением, данные копируются из кэша в переданный буфер.
* Если блока нет:
  + Выделяется выровненный буфер с помощью VirtualAlloc (обязательное условие для работы с FILE\_FLAG\_NO\_BUFFERING).
  + С помощью ReadFile читается блок данных с диска.
  + Создаётся новый элемент кэша с сохранением считанных данных.
  + Если число элементов кэша превышает CACHE\_SIZE, самый старый элемент удаляется.
* Считанные данные копируются в пользовательский буфер, и возвращается число прочитанных байт.

**Функция lab2\_write**

DLL\_EXPORT SSIZE\_T lab2\_write(int fd, const void \*buf, size\_t count) {

if (fd < 0 || fd >= 256 || openFiles[fd] == NULL)

return -1;

FileDescriptor \*file = openFiles[fd];

int fileOffset = lab2\_lseek(fd, 0, SEEK\_CUR);

CacheEntry \*entry = findCacheEntry(file, fileOffset);

if (entry) {

memcpy(entry->data, buf, count);

entry->dirty = 1;

return count;

}

// Выделяем выровненный буфер

char \*alignedBuffer = (char\*)VirtualAlloc(NULL, BLOCK\_SIZE, MEM\_COMMIT | MEM\_RESERVE, PAGE\_READWRITE);

if (!alignedBuffer)

return -1;

memcpy(alignedBuffer, buf, count);

// Создаём новый элемент кэша для записи

CacheEntry \*newEntry = (CacheEntry\*)malloc(sizeof(CacheEntry));

newEntry->fileOffset = fileOffset;

newEntry->data = alignedBuffer;

newEntry->dirty = 1;

newEntry->next = NULL;

if (file->cacheTail)

file->cacheTail->next = newEntry;

file->cacheTail = newEntry;

if (!file->cacheHead)

file->cacheHead = newEntry;

file->cacheCount++;

// Если кэш переполнен, удаляем самый старый элемент,

// предварительно записывая его, если данные изменены

if (file->cacheCount > CACHE\_SIZE) {

CacheEntry \*old = file->cacheHead;

file->cacheHead = old->next;

if (old->dirty) {

DWORD bytesWritten;

LARGE\_INTEGER li;

li.QuadPart = old->fileOffset;

SetFilePointerEx(file->hFile, li, NULL, FILE\_BEGIN);

WriteFile(file->hFile, old->data, BLOCK\_SIZE, &bytesWritten, NULL);

}

VirtualFree(old->data, 0, MEM\_RELEASE);

free(old);

file->cacheCount--;

}

return count;

}

* Проверяется корректность дескриптора.
* Определяется текущая позиция в файле.
* Если элемент кэша с данной позицией найден, обновляются данные и флаг dirty устанавливается в 1.
* Если элемента нет:
  + Выделяется выровненный буфер с помощью VirtualAlloc.
  + Создаётся новый элемент кэша с данными для записи.
  + Новый элемент добавляется в конец списка кэша.
  + Если число элементов превышает CACHE\_SIZE, удаляется самый старый элемент: если он изменён, его содержимое сначала записывается на диск.
* Функция возвращает количество записанных байт.

**Функция lab2\_lseek**

DLL\_EXPORT long lab2\_lseek(int fd, long offset, int whence) {

if (fd < 0 || fd >= 256 || openFiles[fd] == NULL)

return -1;

LARGE\_INTEGER li;

li.QuadPart = offset;

return SetFilePointerEx(openFiles[fd]->hFile, li, NULL, whence) ? li.QuadPart : -1;

}

* Функция устанавливает позицию в файле.
* Принимает смещение (offset) и режим (whence – например, SEEK\_CUR, SEEK\_SET, SEEK\_END).
* Используется функция SetFilePointerEx, которая устанавливает указатель на позицию и возвращает новое смещение.
* Если операция успешна, возвращается новое смещение, иначе – -1.

**Функция lab2\_fsync**

DLL\_EXPORT int lab2\_fsync(int fd) {

if (fd < 0 || fd >= 256 || openFiles[fd] == NULL)

return -1;

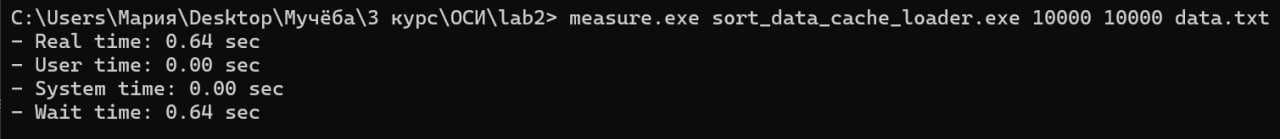
return FlushFileBuffers(openFiles[fd]->hFile) ? 0 : -1;

}

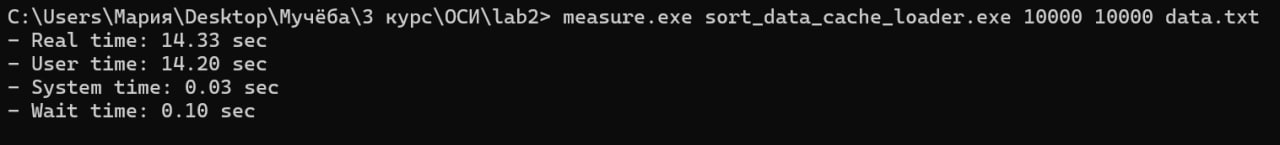
* Функция синхронизирует данные с диском, используя FlushFileBuffers.
* Если данные успешно сброшены на диск, возвращается 0, иначе – -1.

## Данные о работе программы-нагрузчика до и после внедрения

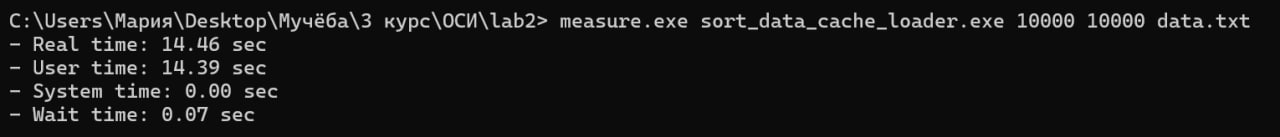
#### С реализованным кэшем



С не отключенным системным кэшем



С отключенным системным кэшем



## Заключение

Наиболее эффективным оказался вариант с реализованным кэшем, чтение и запись выполняются быстрее благодаря предварительному хранению блоков в памяти и минимизации количества обращений к диску.

Механизмы операционной системы не всегда эффективно кэшируют и управляют файлами, особенно при частых чтениях и записях небольшими блоками.

Отключение системного кэша (FILE\_FLAG\_NO\_BUFFERING) также дало плохие результаты, каждое обращение к файлу идёт напрямую на диск без предварительного кэширования.