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

Факультет программной инженерии и компьютерной техники

Дисциплина:

“Операционные системы”

Отчет по лабораторной работе № 2

Работу выполнил:

Лукьянчук Ярослав Евгеньевич

Группа:

P3323

Преподаватель:
Клименков С. В.

Санкт-Петербург

2025 г.

Отчет по лабораторной работе № 2

Репозиторий лабораторной.

Вариант

```
{
  "target_os": "Linux",
  "cache_policy": "LRU"
}
```

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.

Ограничения

- Программа (комплекс программ) должна быть реализован на языке C или C++.
- Запрещено использовать высокоуровневые абстракции над системными вызовами. Необходимо использовать, в случае Unix, процедуры libc.

Краткий обзор кода

1. Общая идея

- `Lab2File` – структура, которая хранит всё необходимое для работы с файлом в вашей библиотеке: настоящий файловый дескриптор, текущий размер файла, «курсор» (смещение для чтения/записи), LRU-список (для управления блоками в кэше) и хеш-таблицу (для ускоренного поиска нужных блоков).
- `CacheBlock` – один блок кэша (по умолчанию 4096 байт). Содержит:
  - `block_number` – номер блока в файле (каждый блок = 4096 байт).
  - `data` – выделенную память под блок.
  - `dirty` – флаг «грязный ли блок» (true, если данные в кэше отличаются от диска).
  - ссылки на блоки в двусвязном LRU-списке и на следующий блок в хеш-цепочке.
- При **чтении** или **записи** данных:
  1. Вычисляется номер блока `block_num = offset / BLOCK_SIZE`.
  2. Ищется блок в хеш-таблице (функция `find_block`).
     - Если блока нет, он загружается (функция `load_block`), при необходимости вытесняя «самый старый» (LRU- `tail`).
  3. Данные копируются в/из блока кэша.
  4. В случае записи блок помечается «грязным» (dirty = true).
- При **закрытии** или явном `lab2_fsync` «грязные» блоки пишутся на диск.

2. Обзор кода

Вспомогательные функции

```
static unsigned hash_off(off_t block_number)
```

code

```
static unsigned hash_off(off_t block_number) {
    return (unsigned)(block_number % CACHE_CAPACITY);
}
```

- **Зачем**: рассчитывает индекс для хеш-таблицы, исходя из номера блока в файле.
- **Как**: берёт номер блока `% CACHE_CAPACITY`, чтобы получить «корзину» (bucket) в хеш-таблице.

`static void move_to_head(Lab2File *f, CacheBlock *b)`

code

```
static void move_to_head(Lab2File *f, CacheBlock *b) {
    if (!b || b == f->lru_head) return;
    if (b->prev) b->prev->next = b->next;
    if (b->next) b->next->prev = b->prev;
    if (f->lru_tail == b) f->lru_tail = b->prev;
    b->prev = NULL;
    b->next = f->lru_head;
    if (f->lru_head) f->lru_head->prev = b;
    f->lru_head = b;
    if (!f->lru_tail) f->lru_tail = b;
}
```

- **Зачем**: если блок уже есть в кэше, при доступе к нему нужно поднять его в голову LRU-списка (он становится «наиболее недавно использованным»).
- **Как**:
  - Удаляет блок из текущего места в двусвязном списке.
  - Ставит его в начало (`lru_head`).

`static void remove_from_hash(Lab2File *f, CacheBlock *b)`

code

```
static void remove_from_hash(Lab2File *f, CacheBlock *b) {
    unsigned i = hash_off(b->block_number);
    CacheBlock *p = f->hash_table[i], *prevp = NULL;
    while (p) {
        if (p == b) {
            if (!prevp) f->hash_table[i] = p->next_hash;
            else prevp->next_hash = p->next_hash;
            return;
        }
        prevp = p;
        p = p->next_hash;
    }
}
```

- **Зачем**: удаляет блок из цепочки хеш-таблицы (когда блок вытесняют или закрывают файл).
- **Как**:
  - Ищет в соответствующей «корзине» (полученной через `hash_off`) блок `b`.
  - Убирает его из связанного списка `next_hash`.

`static CacheBlock* evict_block(Lab2File *f)`

code

```
static CacheBlock* evict_block(Lab2File *f) {
```

```
    CacheBlock *b = f->lru_tail;
    if (!b) return NULL;
    if (b->dirty) {
        off_t off = b->block_number * BLOCK_SIZE;
        pwrite(f->fd, b->data, BLOCK_SIZE, off);
    }
    remove_from_hash(f, b);
    if (b->prev) b->prev->next = NULL;
    f->lru_tail = b->prev;
    if (f->lru_head == b) f->lru_head = NULL;
    f->cache_count--;
    return b;
}
```

- **Зачем**: при переполнении кэша нужно «вытеснить» (удалить) блок. По политике LRU, вытесняем хвост — «самый давно неиспользуемый».
- **Как**:

  1. Берёт `lru_tail` (последний в списке LRU).
  2. Если он «грязный», записывает данные на диск.
  3. Удаляет его из хеш-таблицы и LRU-списка.
  4. Уменьшает счётчик кэша и возвращает указатель на этот блок (чтобы вызывающая функция могла освободить память).

`static CacheBlock* find_block(Lab2File *f, off_t block_num)`

code

```
static CacheBlock* find_block(Lab2File *f, off_t block_num) {
    unsigned i = hash_off(block_num);
    CacheBlock *b = f->hash_table[i];
    while (b) {
        if (b->block_number == block_num) return b;
        b = b->next_hash;
    }
    return NULL;
}
```

- **Зачем**: ищет блок в хеш-таблице (в кэше), чтобы понять, загружен ли уже требуемый блок.
- **Как**:

  - Считает индекс через `hash_off(block_num)`.
  - Проходит по цепочке `next_hash` в этой корзине, сравнивая `block_number`.

`static CacheBlock* load_block(Lab2File *f, off_t block_num)`

code

```
static CacheBlock* load_block(Lab2File *f, off_t block_num) {
    if (f->cache_count >= CACHE_CAPACITY) {
        CacheBlock *victim = evict_block(f);
        if (victim) {
            free(victim->data);
            free(victim);
        }
    }
    CacheBlock *b = malloc(sizeof(CacheBlock));
    posix_memalign((void**)&b->data, BLOCK_SIZE, BLOCK_SIZE);
    b->block_number = block_num;
    b->dirty = false;
```

```
    b->prev = b->next = b->next_hash = NULL;
    {
        off_t off = block_num * BLOCK_SIZE;
        ssize_t r = pread(f->fd, b->data, BLOCK_SIZE, off);
        if (r < 0) memset(b->data, 0, BLOCK_SIZE);
        else if (r < BLOCK_SIZE) memset(b->data + r, 0, BLOCK_SIZE - r);
    }
    {
        unsigned i = hash_off(block_num);
        b->next_hash = f->hash_table[i];
        f->hash_table[i] = b;
    }
    b->next = f->lru_head;
    if (f->lru_head) f->lru_head->prev = b;
    f->lru_head = b;
    if (!f->lru_tail) f->lru_tail = b;
    f->cache_count++;
    return b;
}
```

- **Зачем**: загрузить новый блок из файла в кэш, если он ещё не был загружен.
- **Как**:

> 1. При необходимости (если кэш переполнен) вызывает `evict_block`.
> 2. Выделяет под блок структуру `CacheBlock` и память под `data` (используя `posix_memalign` под прямой ввод-вывод).
> 3. Считывает данные с диска (через `pread`).
> 4. Добавляет блок в начало LRU-списка и в хеш-таблицу.
> 5. Увеличивает счётчик кэша.

Основные интерфейсные функции

`int lab2_open(const char *path)`

code

```
int lab2_open(const char *path) {
    int real_fd = open(path, O_CREAT | O_RDWR | O_DIRECT, 0666);
    if (real_fd < 0) return -1;
    Lab2File *lf = malloc(sizeof(Lab2File));
    memset(lf, 0, sizeof(Lab2File));
    lf->fd = real_fd;
    lf->offset = 0;
    lf->lru_head = NULL;
    lf->lru_tail = NULL;
    lf->cache_count = 0;
    memset(lf->hash_table, 0, sizeof(lf->hash_table));
    lf->file_size = lseek(real_fd, 0, SEEK_END);
    files[file_index] = lf;
    file_index++;
    return file_index - 1;
}

```

- **Зачем**: открывает (или создаёт) реальный файл и инициализирует свою структуру `Lab2File`.
- **Как**:

1. Вызывает `open` с `O_CREAT | O_RDWR | O_DIRECT`.
2. Создаёт `Lab2File`, обнуляет поля (включая кэш и LRU-список).
3. Запоминает полученный `fd` и вычисляет размер файла через `lseek(..., SEEK_END)`.
4. Сохраняет указатель на `Lab2File` в глобальном массиве `files[]`, возвращает индекс.

`int lab2_close(int fd)`

code

```c
int lab2_close(int fd) {
    Lab2File *f = get_file(fd);
    if (!f) return -1;
    for (;;) {
        CacheBlock *b = f->lru_tail;
        if (!b) break;
        if (b->dirty) {
            off_t off = b->block_number * BLOCK_SIZE;
            pwrite(f->fd, b->data, BLOCK_SIZE, off);
        }
        remove_from_hash(f, b);
        if (b->prev) b->prev->next = NULL;
        f->lru_tail = b->prev;
        if (f->lru_head == b) f->lru_head = NULL;
        free(b->data);
        free(b);
    }
    close(f->fd);
    free(f);
    files[fd] = NULL;
    return 0;
}
```

- **Зачем**: закрывает виртуальный дескриптор (и реальный файл), сбрасывает кэш на диск, освобождает память.
- **Как**:
  1. Находит соответствующий `Lab2File` в глобальном массиве (через `get_file`).
  2. Идёт по LRU-списку (от хвоста к голове) и, если блок «грязный», записывает на диск.
  3. Удаляет блоки из хеш-таблицы, освобождает их данные.
  4. Закрывает реальный дескриптор файла (функцией `close`).
  5. Удаляет запись из массива `files[]`.

`ssize_t lab2_read(int fd, void *buf, size_t count)`

code

```c
ssize_t lab2_read(int fd, void *buf, size_t count) {
    Lab2File *f = get_file(fd);
    if (!f) return -1;

    if (f->offset >= f->file_size) {
        return 0;
    }

    if (f->offset + count > f->file_size) {
        count = f->file_size - f->offset;
    }

    size_t total = 0;
```

```
    char *p = buf;
    while (count > 0) {
        off_t bn = f->offset / BLOCK_SIZE;
        size_t off = f->offset % BLOCK_SIZE;
        size_t can_read = BLOCK_SIZE - off;
        if (can_read > count) {
            can_read = count;
        }
        // поиск/загрузка блока
        CacheBlock *b = find_block(f, bn);
        if (!b) {
            b = load_block(f, bn);
        } else {
            move_to_head(f, b);
        }
        // копирование
        memcpy(p, b->data + off, can_read);
        total += can_read;
        p += can_read;
        f->offset += can_read;
        count -= can_read;
    }
    return total;
}
```

- **Зачем**: читает из файла данные, используя кэш (поблочно).
- **Как**:

  1. Находит `Lab2File`.
  2. Проверяет границы (не выходим ли за конец файла).
  3. В цикле пока есть данные для чтения:
     - Вычисляет номер блока (`bn = offset / BLOCK_SIZE`) и смещение в блоке.
     - Пытается найти блок в кэше (`find_block`); если нет — загружает (`load_block`).
     - Копирует нужную часть из кэш-блока в `buf`.
     - Обновляет `offset`, уменьшает `count`.
     - Повторяет до тех пор, пока не прочитается требуемое количество.
  4. Возвращает, сколько байт реально прочитано.

`ssize_t lab2_write(int fd, const void *buf, size_t count)`

code

```
ssize_t lab2_write(int fd, const void *buf, size_t count) {
    Lab2File *f = get_file(fd);
    if (!f) return -1;
    size_t total = 0;
    const char *p = buf;
    while (count > 0) {
        off_t bn = f->offset / BLOCK_SIZE;
        size_t off = f->offset % BLOCK_SIZE;
        size_t can_write = BLOCK_SIZE - off;
        if (can_write > count) can_write = count;
        CacheBlock *b = find_block(f, bn);
        if (!b) {
            if (off != 0 || can_write < BLOCK_SIZE) b = load_block(f, bn);
            else {
                if (f->cache_count >= CACHE_CAPACITY) {
                    CacheBlock *victim = evict_block(f);
```

```
                    if (victim) {
                        free(victim->data);
                        free(victim);
                    }
                }
                b = malloc(sizeof(CacheBlock));
                posix_memalign((void**)&b->data, BLOCK_SIZE, BLOCK_SIZE);
                memset(b->data, 0, BLOCK_SIZE);
                b->block_number = bn;
                b->dirty = false;
                b->prev = b->next = b->next_hash = NULL;
                {
                    unsigned i = hash_off(bn);
                    b->next_hash = f->hash_table[i];
                    f->hash_table[i] = b;
                }
                b->next = f->lru_head;
                if (f->lru_head) f->lru_head->prev = b;
                f->lru_head = b;
                if (!f->lru_tail) f->lru_tail = b;
                f->cache_count++;
            }
        } else move_to_head(f, b);
        memcpy(b->data + off, p, can_write);
        b->dirty = true;
        total += can_write;
        p += can_write;
        f->offset += can_write;
        if (f->offset > f->file_size) f->file_size = f->offset;
        count -= can_write;
        if (f->cache_count > CACHE_CAPACITY) {
            CacheBlock *victim = evict_block(f);
            if (victim) {
                free(victim->data);
                free(victim);
            }
        }
    }
    return total;
}
```

- **Зачем**: записывает данные, используя кэш (поблочно).
- **Как**:

> 1. Находит `Lab2File`.
> 2. В цикле разбивает `count` на части по размеру кэш-блока (4096 байт) с учётом внутреннего смещения в блоке.
> 3. Ищет блок в кэше. Если отсутствует, загружает (если нужно частично обновить блок) или создаёт новый пустой блок (если перекрывается весь 4096).
> 4. Копирует данные из пользовательского буфера в `b->data`.
> 5. Ставит `b->dirty = true`.
> 6. Двигает `offset` вперёд, обновляет `file_size`, если ушли дальше «конца».
> 7. При переполнении кэша вызывает `evict_block`.
> 8. Возвращает, сколько байт записано.

`off_t lab2_lseek(int fd, off_t offset, int whence)`

▾ code

```
off_t lab2_lseek(int fd, off_t offset, int whence) {
    Lab2File *f = get_file(fd);
    if (!f) return -1;
    off_t new_off;
    if (whence == SEEK_SET) new_off = offset;
    else if (whence == SEEK_CUR) new_off = f->offset + offset;
    else if (whence == SEEK_END) new_off = f->file_size + offset;
    else return -1;
    if (new_off < 0) return -1;
    f->offset = new_off;
    return f->offset;
}
```

- **Зачем**: меняет «курсор» (текущее смещение в файле).
- **Как**:

  1. Находит `Lab2File`.
  2. Вычисляет новый `offset` в зависимости от `whence` (`SEEK_SET`, `SEEK_CUR`, `SEEK_END`).
  3. Запоминает его в структуре `Lab2File` (если не уходит в «отрицательное» значение).
  4. Возвращает текущий `offset`.

`int lab2_fsync(int fd)`

code

```
int lab2_fsync(int fd) {
    Lab2File *f = get_file(fd);
    if (!f) return -1;
    CacheBlock *b = f->lru_head;
    while (b) {
        if (b->dirty) {
            off_t off = b->block_number * BLOCK_SIZE;
            pwrite(f->fd, b->data, BLOCK_SIZE, off);
            b->dirty = false;
        }
        b = b->next;
    }
    fsync(f->fd);
    return 0;
}

```

- **Зачем**: сбрасывает все «грязные» (dirty) блоки на диск, чтобы гарантировать сохранение.
- **Как**:

  1. Находит `Lab2File`.
  2. Проходит по всему LRU-списку (от `lru_head` к `lru_tail`).
  3. Если блок «грязный», выполняет `pwrite` и сбрасывает флаг `dirty`.
  4. Вызвает `fsync` на реальном `fd`.
  5. Возвращает 0 при успехе (или -1 при ошибке).

Результаты тестов

results

```
       _,met$$$$$gg.          debian@debian
```

```
    ,g$$$$$$$$$$$$$$$P.       -------------
  ,g$$P"     """Y$$.".        OS: Debian GNU/Linux 12 (bookworm) aarch64
 ,$$P'              `$$$.     Host: QEMU Virtual Machine virt-7.2
',$$P       ,ggs.     `$$b:   Kernel: 6.1.0-28-arm64
`d$$'     ,$P"'   .    $$$    Uptime: 1 hour, 49 mins
 $$P      d$'     ,    $$P    Packages: 1670 (dpkg)
 $$:      $$.   -    ,d$$'    Shell: bash 5.2.15
 $$;      Y$b._   _,d$P'      Resolution: 1800x1126
 Y$$.    `.`"Y$$$$P"'         DE: GNOME 43.9
 `$$b      "-.__              WM: Mutter
  `Y$$                        WM Theme: Adwaita
   `Y$$.                      Theme: Adwaita [GTK2/3]
     `$$b.                    Icons: Adwaita [GTK2/3]
       `Y$$b.                 Terminal: vscode
          `"Y$b._             CPU: (6)
              `"""            GPU: 00:02.0 Red Hat, Inc. Virtio 1.0 GPU
                              Memory: 1750MiB / 3921MiB

==================================================
              Performance Test Suite
==================================================

Test 1: LRU Cache Performance Test
Description: Evaluating cache performance with different
file sizes and access patterns (sequential and random)
==================================================

Size(MB) | Mode  | Run | no_cache(ms) | with_cache(ms)
-------------------------------------------------------
256      | seq |  1  | 556.99       | 121.61
256      | seq |  2  | 180.57       | 82.68
256      | seq |  3  | 144.44       | 97.06
256      | rand |  1  | 131.31       | 63.39
256      | rand |  2  | 107.42       | 74.26
256      | rand |  3  | 184.01       | 84.68
512      | seq |  1  | 269.37       | 145.71
512      | seq |  2  | 262.25       | 144.72
512      | seq |  3  | 213.70       | 128.11
512      | rand |  1  | 185.56       | 123.67
512      | rand |  2  | 187.56       | 123.31
512      | rand |  3  | 185.75       | 126.48
1024      | seq |  1  | 620.01       | 1760.32
1024      | seq |  2  | 553.51       | 1338.44
1024      | seq |  3  | 508.06       | 1708.58
1024      | rand |  1  | 377.81       | 1088.60
1024      | rand |  2  | 401.16       | 1119.03
1024      | rand |  3  | 365.50       | 1095.07

==================================================
Test 2: External Integer Sorting Test
Description: Testing the performance of external
merge sort implementation for integer arrays
==================================================
 total_ints | chunk_size |   sys_time(ms)  |  lab2_time(ms)
------------+------------+-----------------+----------------
      20000 |       2000 |           57.36 |           5.99
      50000 |       5000 |          140.39 |          12.77
     100000 |      10000 |          254.05 |          27.84
```

Анализ результатов

1. **Файлы 256MB и 512MB**

- При меньших объёмах файла использование кэша даёт заметное преимущество: время «with_cache» почти всегда ниже, чем «no_cache», особенно при повторных последовательных доступах.
- В режиме `random` (случайный доступ) кэш даёт ещё больший выигрыш: число «хитовых» обращений в кэш растёт, что экономит обращения к диску.

2. **Файлы 1024MB (1GB)**

   - На объёме 1GB картина меняется: «with_cache» показал время **выше**, чем «no_cache».
   - Возможные причины:
     - Для больших объёмов данных возрастают накладные расходы на пользовательский кэш (управление структурами кэша, пересылка данных из/в буфер), особенно учитывая, что в коде используется `O_DIRECT`, и приходилось читать/записывать через выделенные выравненные буферы.
     - При последовательном чтении больших объёмов пользовательский LRU-кэш может сработать хуже, чем встроенный механизмы ядра (page cache), либо при включённом `O_DIRECT` «преимущества» кэширования частично теряются.
     - Возможен дополнительный overhead при больших block-номерах и некоторых особенностях реализации.

   Таким образом, для крупной линейной обработки больших файлов прямое чтение (без пользовательского кэша) иногда оказывается быстрее, поскольку мы фактически дублируем логику ОС, но с дополнительной затратой ресурсов в пространстве пользователя.

3. **Тест внешней сортировки (External Integer Sorting)**

   - Здесь, наоборот, «lab2_time» значительно меньше, чем «sys_time». То есть использование кэша при внешней сортировке принесло пользу.
   - Внешняя сортировка активно работает с данным файлом «кусками» (chunk), и при повторных обращениях те же блоки данных часто уже находятся в кэше. Кроме того, запись «грязных» блоков происходит реже, чем мелкие «прямые» записи без кэша.
   - В итоге время выполнения заметно сокращается благодаря уменьшению количества прямых обращений к диску.

Выводы

- При небольших и средних объёмах файлов (до ~512 MB) и многократном доступе к одним и тем же блокам реализация LRU-кэша значительно ускоряет операции ввода-вывода.
- При больших файлах (1 GB и более) преимущество может снижаться или даже приводить к ухудшению производительности из-за накладных расходов на работу кэша в пространстве пользователя.
- Внешняя сортировка получает заметный выигрыш за счёт снижения количества прямых обращений к диску, что подтверждает эффективность кэша для сценариев, где блоки многократно переиспользуются.