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

Факультет программной инженерии и компьютерной техники

**Отчёт**

**по лабораторной работе №2**

по дисциплине «Операционные системы»

Вариант: *Linux FIFO*

Выполнил:

Рудкевич И. А., группа P3306

Преподаватель:

Тюрин И. Н.

Санкт-Петербург

~ 2024 ~

Оглавление

[Задание 3](#_Toc184208921)

[Листинг исходного кода 4](#_Toc184208922)

[Измерения 10](#_Toc184208923)

[Заключение 10](#_Toc184208924)

[Вывод 10](#_Toc184208925)

# Задание

В данной лабораторной работе необходимо реализовать блочный кэш в пространстве пользователя в виде динамической библиотеки (.so). Политику вытеснения: Linux FIFO

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

* Открытие файла по заданному пути файла, доступного для чтения. Процедура возвращает некоторый хэндл на файл. Пример: int lab2\_open(const char \*path)
* Закрытие файла по хэндлу. Пример: int lab2\_close(int fd)
* Чтение данных из файла. Пример: ssize\_t lab2\_read(int fd, void buf[.count], size\_t count)
* Запись данных в файл. Пример: ssize\_t lab2\_write(int fd, const void buf[.count], size\_t count)
* Перестановка позиции указателя на данные файла. Достаточно поддержать только абсолютные координаты. Пример: off\_t lab2\_lseek(int fd, off\_t offset, int whence)
* 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 программу-загрузчик из ЛР 1, добавив использование кэша. Запустить программу и убедиться, что она корректно работает. Сравнить производительность до и после.

# Листинг исходного кода

**lib/src/cache.c**

#include "../include/cache.h"

#include <errno.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

Cache cache = {NULL, NULL, 0};

CacheBlock \*find\_cache\_block(int fd, off\_t offset) {

CacheBlock \*block = cache.head;

while (block != NULL) {

if (block->fd == fd && block->offset == offset) {

return block;

}

block = block->next;

}

return NULL;

}

void add\_cache\_block(int fd, off\_t offset, const char \*data) {

// check if cache is full

if (cache.current\_size >= CACHE\_SIZE) {

CacheBlock \*old\_block = cache.head;

if (old\_block) {

// write the data of the oldest block back to disk to persist any changes

pwrite(old\_block->fd, old\_block->data, BLOCK\_SIZE, old\_block->offset);

// remove the oldest block

cache.head = old\_block->next;

if (cache.head)

cache.head->prev = NULL;

if (cache.tail == old\_block)

cache.tail = NULL;

free(old\_block);

cache.current\_size--;

}

}

// create and add the new block to the tail (most recent)

CacheBlock \*new\_block = (CacheBlock \*)malloc(sizeof(CacheBlock));

new\_block->fd = fd;

new\_block->offset = offset;

memcpy(new\_block->data, data, BLOCK\_SIZE);

new\_block->next = NULL;

new\_block->prev = cache.tail;

if (cache.tail)

cache.tail->next = new\_block;

cache.tail = new\_block;

if (cache.head == NULL)

cache.head = new\_block;

cache.current\_size++;

}

ssize\_t read\_block(int fd, off\_t offset, char \*buffer) {

// look for the requested block in the cache

CacheBlock \*block = find\_cache\_block(fd, offset);

if (block != NULL) {

// if the block is found in the cache, copy its data to the buffer

memcpy(buffer, block->data, BLOCK\_SIZE);

return BLOCK\_SIZE;

} else {

// if the block is not found in the cache,

// read from disk (and save to cache)

ssize\_t bytes\_read = pread(fd, buffer, BLOCK\_SIZE, offset);

if (bytes\_read > 0)

add\_cache\_block(fd, offset, buffer); // add block to the cache

return bytes\_read;

}

}

ssize\_t write\_block(int fd, off\_t offset, const char \*data) {

// look for the requested block in the cache

CacheBlock \*block = find\_cache\_block(fd, offset);

if (block != NULL) {

// if the block is found in the cache, update its data

memcpy(block->data, data, BLOCK\_SIZE);

return BLOCK\_SIZE;

} else {

// add to cache, if the block is not found

add\_cache\_block(fd, offset, data);

return BLOCK\_SIZE;

}

}

**lib/src/api.c**

#include "../include/api.h"

#include "../include/cache.h"

#include <errno.h>

#include <fcntl.h>

#include <stdlib.h>

#include <string.h>

#include <sys/fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

int c\_open(const char \*path) {

int flags = O\_RDWR | O\_SYNC;

#if defined(\_\_APPLE\_\_)

flags |= F\_NOCACHE;

#elif defined(\_\_linux\_\_)

flags |= O\_DIRECT;

#endif

// NOTE: F\_NOCACHE for macos, O\_DIRECT for linux

return open(path, flags);

}

int c\_close(int fd) {

// remove fd from cache before closing it

CacheBlock \*block = cache.head;

while (block != NULL) {

if (block->fd == fd) {

// if the block is at the head of the cache

if (block == cache.head) {

cache.head = block->next;

if (cache.head)

cache.head->prev = NULL;

}

// if the block is at the tail of the cache

if (block == cache.tail) {

cache.tail = block->prev;

if (cache.tail)

cache.tail->next = NULL;

}

// if the block is somewhere in the middle

if (block->prev != NULL) {

block->prev->next = block->next;

}

if (block->next != NULL) {

block->next->prev = block->prev;

}

free(block);

cache.current\_size--;

break;

}

block = block->next;

}

return close(fd);

}

off\_t align\_offset(off\_t offset) {

return offset & ~(BLOCK\_SIZE - 1);

}

ssize\_t c\_read(int fd, void \*buf, size\_t count) {

off\_t offset = lseek(fd, 0, SEEK\_CUR);

ssize\_t bytes\_read = 0;

size\_t remaining = count;

while (remaining > 0) {

// align the offset to the block boundary

off\_t block\_offset = align\_offset(offset);

char block[BLOCK\_SIZE];

ssize\_t bytes\_in\_block = read\_block(fd, block\_offset, block);

if (bytes\_in\_block <= 0)

return bytes\_in\_block;

// copy data from the block to the buffer (either remaining or block size)

size\_t to\_copy = remaining < bytes\_in\_block ? remaining : bytes\_in\_block;

memcpy(buf + bytes\_read, block + (offset - block\_offset), to\_copy);

bytes\_read += to\_copy;

remaining -= to\_copy;

offset += to\_copy;

}

return bytes\_read;

}

ssize\_t c\_write(int fd, const void \*buf, size\_t count) {

off\_t offset = lseek(fd, 0, SEEK\_CUR);

ssize\_t bytes\_written = 0;

size\_t remaining = count;

while (remaining > 0) {

// align the offset to the block boundary

off\_t block\_offset = align\_offset(offset);

char block[BLOCK\_SIZE];

// copy data into block buffer (either remaining or full block size)

size\_t to\_copy = remaining < BLOCK\_SIZE ? remaining : BLOCK\_SIZE;

memcpy(block, buf + bytes\_written, to\_copy);

ssize\_t written = write\_block(fd, block\_offset, block);

if (written <= 0)

return written;

bytes\_written += to\_copy;

remaining -= to\_copy;

offset += to\_copy;

}

return bytes\_written;

}

int c\_fsync(int fd) {

// write all blocks back to disk

CacheBlock \*block = cache.head;

while (block != NULL) {

pwrite(block->fd, block->data, BLOCK\_SIZE, block->offset);

block = block->next;

}

return fsync(fd);

}

off\_t c\_lseek(int fd, off\_t offset, int whence) {

return lseek(fd, offset, whence);

}

**src/ema\_search\_with\_cache.c**

#include "../lib/include/api.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <unistd.h>

#define BUFFER\_SIZE 128

int search\_substring(const char \*buffer, const char \*substring) {

int i, j;

if (substring[0] == '\0') {

return -1;

}

for (i = 0; buffer[i] != '\0'; i++) {

for (j = 0; substring[j] != '\0'; j++) {

if (buffer[i + j] != substring[j]) {

break;

}

}

if (substring[j] == '\0') {

return i;

}

}

return -1;

}

int process\_file(int fd, const char \*substring) {

off\_t offset = 0;

int iteration = 0;

char buffer[BUFFER\_SIZE];

while (1) {

if (c\_lseek(fd, offset, SEEK\_SET) == (off\_t)-1) {

perror("Error seeking file");

return EXIT\_FAILURE;

}

ssize\_t bytes\_read = c\_read(fd, buffer, BUFFER\_SIZE);

if (bytes\_read <= 0) {

break;

}

int index = search\_substring(buffer, substring);

if (index != -1) {

printf("Found substring at position %lld\n", offset + index);

break;

}

printf("Not Found substring at offset %lld\n", offset);

iteration++;

offset += bytes\_read;

}

return EXIT\_SUCCESS;

}

int main(int argc, char \*argv[]) {

if (argc < 4) {

fprintf(stderr, "Usage: %s <filename> <substring> <repeats>\n", argv[0]);

return EXIT\_FAILURE;

}

const char \*filename = argv[1];

const char \*substring = argv[2];

size\_t repeats = atoi(argv[3]);

if (strlen(substring) > BUFFER\_SIZE) {

fprintf(stderr, "Substring is larger than buffer size\n");

return EXIT\_FAILURE;

}

int fd = c\_open(filename);

if (fd == -1) {

perror("Error opening file");

return EXIT\_FAILURE;

}

clock\_t total\_start = clock();

for (int r = 0; r < repeats; r++) {

clock\_t start = clock();

if (process\_file(fd, substring) != 0) {

fprintf(stderr, "[%d] AHTUNG AHTUNG SOME ERROR OCCURRED\n", r);

}

printf("Execution time for repetition %d: %lf seconds\n\n", r,

(double)(clock() - start) / CLOCKS\_PER\_SEC);

}

clock\_t total\_end = clock();

c\_close(fd); // Close the file using c\_close instead of fclose

printf("\n");

printf(">>> Total execution time: %lf seconds <<<\n\n",

(double)(total\_end - total\_start) / CLOCKS\_PER\_SEC);

return EXIT\_SUCCESS;

}

# Измерения

Запуск программы-нагрузчика ema\_search **без** использования разработанного кэша (одна итерация):

>>> Total execution time: 1.058498 seconds <<<

Запуск программы-нагрузчика ema\_search **с** использованием разработанного кэша (одна итерация):

>>> Total execution time: 9.641296 seconds <<<

## Заключение

Как можно заметить, использование разработанного кэша замедлило выполнение программы-нагрузчика в 9 раз. Такая деградация в производительности может быть обусловлена следующими факторами:

1. Политика вытеснения FIFO
2. Программа-нагрузчик "поиск подстроки во внешней памяти" читает данные по чанкам из внешней памяти. Если в текущем чанке подстрока не найдена, программа переходит к следующему. При этом к предыдущим чанкам повторного обращения нет, что ведёт к постоянным кэш-промахам.

# Вывод

В ходе выполнения лабораторной работы я ознакомился с политикой вытеснения Linux FIFO, узнал как можно осуществлять операции с диском в обход page cache на Linux и MacOS. А также научился собирать проект в виде динамической библиотеки.