Лабораторная работа №2

Выполнил: Трошкин Александр Евгеньевич, 333304
Группа: P3316
Преподаватель: Гиниятуллин Арслан Рафаилович
Вариант: NRU Cache

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++.
- 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.

Обзор кода

Для реализации лабораторной работы мне было необходимо реализовать NRU Cache для работы с внешней памятью, а потом проверить скорость его работы через реализованную программу-нагрузчик, где можно будет наглядно увидеть разницу в скорости работы и сделать по ней выводы.

Реализацию NRU cache библиотеки можно посмотреть тут

https://github.com/k1nd-cat/os-labs/tree/main/lab2_2

Структура файлов:

```
cache/
├── CMakeLists.txt
├── app/
│   └── main.cpp
├── include/
│   └── file_operations.h
│   └── nru_cache.h
└── src/
    └── file_operations.cpp
    └── nru_cache.cpp
```

Для сравнения работы кеша, были написаны следующие алгоритмы:

- **RandomRead_Cached**: Тестирует случайное чтение с использованием кэширования.
- **RandomRead_Uncached**: Тестирует случайное чтение без использования кэширования.
- **MixedWorkload_Cached**: Тестирует смешанную нагрузку (70% чтения, 30% записи) с использованием кэширования.
- **MixedWorkload_Uncached**: Тестирует смешанную нагрузку без использования кэширования.
- **TightAreaRandomRead_Cached**: Тестирует случайное чтение в "горячей" области файла с использованием кэширования.
- **TightAreaRandomRead_Uncached**: Тестирует случайное чтение в "горячей" области файла без использования кэширования.
- **SequentialRead_Cached**: Тестирует последовательное чтение с использованием кэширования.
- **SequentialRead_Uncached**: Тестирует последовательное чтение без использования кэширования.

Результат выполнения алгоритмов

```
=== Random Read Tests ===
RandomRead_Cached:   33.82 ms
RandomRead_Uncached: 4.78 ms
-------------------------------------------------

=== Mixed Workload Tests ===
MixedWorkload_Cached:   43.25 ms
MixedWorkload_Uncached: 618.19 ms
-------------------------------------------------

=== Tight Area Random Read Tests ===
TightAreaRandomRead_Cached:   73.64 ms
TightAreaRandomRead_Uncached: 205.62 ms
-------------------------------------------------

=== Sequential Read Tests ===
SequentialRead_Cached:   980.44 ms
SequentialRead_Uncached: 344.34 ms
-------------------------------------------------
```

Оценка результатов

- Random Read: С кешированием выполняется в 7 раз медленнее, вероятнее всего из-за частых кеш промахов и накладных расходов
- Mixed Workload: Кэш значительно снизил время выполнения теста (в 14 раз), поскольку буферизация памяти и чтение из памяти снизили количество обращений к диску
- Tight Area Random Read: Кеширование значительно ускорило работу (в ~2.8 раза)
- Sequential Read: В последовательном чтении работа кеша значительно больше (в 2.8 раза), поскольку кеш не может эффективно кешировать данные и замедляет выполнение из-за накладных расходов.

Вывод

В данной лабораторной работе я написал алгоритм кеширования NRU и исследовал его работу на практике.

Алгоритм вытеснения NRU заключается в том, что блок на вытеснение определяется в зависимости от его приоритета:

- Класс 0: `!Accessed && !Dirty` — блок не использовался и не изменен.
- Класс 1: `!Accessed && Dirty` — блок не использовался, но был изменен.
- Класс 2: `Accessed && !Dirty` — блок использовался, но не изменен.
- Класс 3: `Accessed && Dirty` — блок использовался и был изменен.

Данный кеш будет неэффективен для чтения больших данных и для последовательного чтения.