Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Курсова робота

з дисципліни “Системне програмне забезпечення”

на тему “Алгоритми планування введення-виведення для жорсткого диска та управління буферним кешем”

Виконав:

студент групи ІО-13

Мальований Д. О.

Перевірив:

Череватенко О. В.

Київ 2024

Опис реалізованих алгоритмів планування введення-виведення для жорсткого диска, алгоритму управління буферним кешем та програми

Варіант завдання: 1315 % 4 + 1 = 4 (LRU з двома сегментами, FIFO, LOOK, FLOOK).

Алгоритм FIFO – це алгоритм планування введення-виведення для жорсткого диска, що заснований на черзі. Таким чином запити виконуються у порядку надходження. Такий алгоритм є найпростішим, проте не враховує можливість об’єднання запитів за номером доріжки для зменшення часу, витраченого на переміщення магнітної голівки диску.

Алгоритм LOOK – це алгоритм планування введення-виведення для жорсткого диска, де запити сортуються за номером доріжки та виконуються у певному напрямку. Максимальна кількість звернень до однієї доріжки обмежена. Такий алгоритм вирішує проблему зменшення часу, витраченого на переміщення магнітної голівки диску, проте не враховує порядок надходження запитів.

Алгоритм FLOOK – це алгоритм планування введення-виведення для жорсткого диска, що вирішує одразу дві проблеми. Існує дві черги: поки виконуються запити з першої черги, запити додаються у другу чергу, і навпаки. Таким чином у середньому запити, що надійшли раніше, виконаються раніше. Проте кожна окрема черга працює подібно до алгоритму LOOK, що дозволяє зменшити часу, витрачений на переміщення магнітної голівки диску.

Алгоритм LRU – це алгоритм управління буферним кешем, що ставить у пріоритет буфери, до яких нещодавно було здійснено доступ. Такий кеш складається з двох сегментів (списків буферів): лівого та правого. Новий буфер додається до початку лівого сегмента. Якщо місця не вистачає, видаляється буфер з кінця лівого сегмента. Коли відбувається доступ до буфера, що є у лівому сегменті, він переміщується на початок правого. Таким чином у правому

сегменті знаходяться буфери, до який нещодавно було здійснено хоча б два доступи.

Опис роботи програми

Програма складається з трьох основних класів:

1. KrScheduler – планувальник процесів;
2. KrBufferCache – буферний кеш;
3. KrDriver – драйвер жорсткого диску;

також є структури, що зберігають допоміжну інформацію:

4. KrUserProcess – процес користувача;
5. KrIORequest – запит на операцію введення-виведення;
6. KrBuffer – буфер у буферному кеші;

і типи переліченння:

7. KrIOOperationType – тип операції: введення або виведення;
8. KrIORequestState – стан процесу користувача та операції введення-виведення: до запиту, запит, заблоковано запитом, не заблоковано запитом.

KrDriver є абстрактним класом, його наслідують класи KrDriverFIFO, KrDriverLOOK та KrDriverFLOOK, що реалізовують відповідні алгоритми.

Хоча клас KrBufferCache можна було зробити абстрактним і наслідувати KrBufferCacheLRU від нього, у курсовій роботі потрібно реалізувати лише одну стратегію буферного кешу, тому для простоти він не є абстрактним і одразу застосовує алгоритм LRU.

Усі типи мають префікс Kr для позначення приналежності до проєкту.

У функції main створюються об’єкти цих класів, додаються процеси користувачів до планувальника процесів, та викликається функція KrIOScheduler::Tick у циклі, що завершується, коли функція повертає false.

Усередині KrIOScheduler::Tick обираємо оброблюємо вибраний процес користувача, опираючись на стан першого його запиту на IO. Якщо процес більше не має запитів, завершуємо його і переходимо до наступного. Якщо вибраний процес заблоковано операцією IO, переходимо до наступного.

Наступний процес шукаємо по колу, поки не знайдемо такий, що не заблокований операцією IO. Якщо усі процеси завершилися або заблоковані, перевіряємо, чи заплановано переривання. Якщо заплановані – переходимо до необхідного моменту часу та виконуємо. Якщо ні – очищаємо буферний кеш.

Коли оброблюємо процес, враховуємо заплановані переривання, і виконуємо їх, якщо необхідно, запам’ятовуючи час, який ми витратили на процес користувача, аби потім продовжити його з моменту переривання.

Під час обробки процесу користувача, якщо необхідно здійснити операцію введення-виведення, звертаємося до буферного кешу. Якщо необхідний буфер присутній у кеші, не блокуємо процес і продовжуємо його виконання. Якщо процес модифікує буфер, позначаємо буфер як модифікований. Якщо потрібний буфер не знайдено, відправляємо драйверу диску запит на читання. Якщо місця для нового буферу недостатньо, видаляємо певний буфер відповідно до вибраного стратегії та відправляємо запит на запис на диск, якщо буфер було модифіковано.

Коли необхідно очистити буферний кеш, видаляємо усі буфери, і для таких, що були модифіковані з моменту читання, відправляємо запити на запис до жорсткого диску.

Коли драйвер диску отримує запит на введення-виведення, додаємо його у до черги відповідно до стратегії, а також починаємо виконувати його, якщо більше нема запланованих запитів. Коли запит виконано, генеруємо переривання, додаємо буфер до кешу, розблоковуємо процес користувача, що надіслав цей запит, а також усі інші, які також надсилали запит на читання цього сектору. У такому випадку видаляємо їх запити з черги. Коли процеси

розблоковано, вони можуть одразу модифікувати буфер. Потім переходимо до виконання наступного запиту, якщо такий є.

Коли необхідно виконати запит на введення-виведення, драйвер диску визначає найшвидший спосіб доступу до потрібного сектора, і планує переривання у KrScheduler.

Висновки про розробленні алгоритми

Алгоритм FIFO – це найпростішим у реалізації алгоритм планування введення-виведення для жорсткого диска, проте найповільніший у випадках, коли сусідні запити звертаються до кількох різних доріжок почергово (наприклад, 1-2-1-2 або 1-2-3-1-2-3).

Алгоритм LOOK виправляє цю проблему, сортуючи запити за номером сектора та виконуючи декілька послідовних звернень до однієї доріжки. Хоча це дозволяє зменшити час, витрачений на переміщення магнітної головки, така стратегія не враховує порядок надходження запитів. Для найкращої продуктивності роботи максимальну кількість послідовних звернень до однієї доріжки потрібно налаштовувати відповідно до конкретної системи.

Алгоритм FLOOK є рішенням обох проблем (а радше компромісом), адже він працює подібно до LOOK, але нові запити не потрапляють в активну чергу. Таким чином більш ранні запити в середньому мають пріоритет над більш пізніми.

Алгоритм LRU ставить у пріоритет ті буфери, до яких зверталися нещодавно, а отже припускає, що саме до них будуть звертатися незабаром. Такий підхід значно кращий за алгоритм, що заснований на черзі, проте має проблему, при якій буфер, до якого було здійснено два послідовні доступи, у пріоритеті над тим, до якого було здійснено один доступ, але будуть багаторазово звертатися у майбутньому). Для вирішення такої проблеми застосовується алгоритм LFU.

Під час виконання курсової роботи я дослідив будову та механізм роботи жорсткого диску та буферного кешу, розробив симулятор багатозадачної системи, процеси у якій звертаються до жорсткого диску через буферний кеш, а також закріпив навички з розробки системного програмного забезпечення.

Лістинг програми

main.cpp

```
/*
 * Системне програмне забезпечення
 * Курсова робота
 * Алгоритми планування введення-виведення для жорсткого
диска та управління буферним кешем
 * Мальований Денис Олегович, ІО-13
 * Варіант 4 (1315 % 4 + 1 = 4)
 * LRU з двома сегментами, FIFO, LOOK, FLOOK
 */

#include "BufferCache.h"
#include "DriverFIFO.h"
#include "DriverFLOOK.h"
#include "DriverLOOK.h"
#include "Scheduler.h"

#include <iostream>

int main()
{
    int ExampleIndex;
    std::cout << "Enter example index (0-8):";
    std::cin >> ExampleIndex;

    int DiskDriverStrategy;
    std::cout << "Enter disk driver strategy (1-3):";
    std::cin >> DiskDriverStrategy;
```

```
    std::cout << "\n";

    KrScheduler Scheduler;
    KrBufferCache BufferCache;
    KrDriver* Driver = nullptr;
    if (DiskDriverStrategy == 1)
    {
        Driver = new KrDriverFIFO{};
    }
    else if (DiskDriverStrategy == 2)
    {
        Driver = new KrDriverLOOK{};
    }
    else if (DiskDriverStrategy == 3)
    {
        Driver = new KrDriverFLOOK{};
    }
    else
    {
        std::cout << "CONSOLE: Unsupported strategy
index\n";
        return 0;
    }

    std::cout << "CONSOLE: Change
MaxConsecutiveAccessToTrackNum in the header file\n\n";
    std::cout << "CONSOLE: Change BufferNum and
SegmentRightBufferNum in the header file\n\n";

    Scheduler.SetBufferCache(&BufferCache);
```

```
    Scheduler.SetDriver(Driver);
    BufferCache.SetDriver(Driver);
    Driver->SetScheduler(&Scheduler);
    Driver->SetBufferCache(&BufferCache);

    std::cout << "Settings:\n";
    Scheduler.PrintSettings();
    BufferCache.PrintSettings();
    Driver->PrintSettings();
    std::cout << "\n";

    // Example #1
    if (ExampleIndex == 1)
    {
        KrUserProcess UserProcess0;
        UserProcess0.Name = "yyy";

UserProcess0.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Read});
        Scheduler.Enqueue(UserProcess0);
    }

    // Example #2
    if (ExampleIndex == 2)
    {
        KrUserProcess UserProcess0;
        UserProcess0.Name = "yyy";

UserProcess0.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Write});
```

```
        Scheduler.Enqueue(UserProcess0);
    }

    // Example #3
    if (ExampleIndex == 3)
    {
        KrUserProcess UserProcess0;
        UserProcess0.Name = "yyy";

UserProcess0.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Read});
        Scheduler.Enqueue(UserProcess0);

        KrUserProcess UserProcess1;
        UserProcess1.Name = "qqq";

UserProcess1.IORequests.push_back(KrIORequest{1000,
KrIOOperationType::Write});
        Scheduler.Enqueue(UserProcess1);
    }

    // Example #4
    if (ExampleIndex == 4)
    {
        KrUserProcess UserProcess0;
        UserProcess0.Name = "yyy";

UserProcess0.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Read});
        Scheduler.Enqueue(UserProcess0);
```

```
        KrUserProcess UserProcess1;
        UserProcess1.Name = "qqq";

UserProcess1.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Read});
        Scheduler.Enqueue(UserProcess1);
    }

    // Example #5
    if (ExampleIndex == 5)
    {
        KrUserProcess UserProcess0;
        UserProcess0.Name = "yyy";

UserProcess0.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Write});

UserProcess0.IORequests.push_back(KrIORequest{110,
KrIOOperationType::Read});

UserProcess0.IORequests.push_back(KrIORequest{120,
KrIOOperationType::Read});

UserProcess0.IORequests.push_back(KrIORequest{130,
KrIOOperationType::Read});

UserProcess0.IORequests.push_back(KrIORequest{140,
KrIOOperationType::Read});
```

```
UserProcess0.IORequests.push_back(KrIORequest{150,
KrIOOperationType::Read});

UserProcess0.IORequests.push_back(KrIORequest{160,
KrIOOperationType::Read});

UserProcess0.IORequests.push_back(KrIORequest{170,
KrIOOperationType::Write});
        Scheduler.Enqueue(UserProcess0);
    }

    // Example #6
    if (ExampleIndex == 6)
    {
        KrUserProcess UserProcess0;
        UserProcess0.Name = "yyy";

UserProcess0.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Write});
        Scheduler.Enqueue(UserProcess0);

        KrUserProcess UserProcess1;
        UserProcess1.Name = "qqq";

UserProcess1.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Read});
        Scheduler.Enqueue(UserProcess1);
    }

    // Example #7 & #8
```

```
    if (ExampleIndex == 7 || ExampleIndex == 8)
    {
        KrUserProcess UserProcess0;
        UserProcess0.Name = "yyy";

UserProcess0.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Read});
        Scheduler.Enqueue(UserProcess0);

        KrUserProcess UserProcess1;
        UserProcess1.Name = "qqq";

UserProcess1.IORequests.push_back(KrIORequest{110,
KrIOOperationType::Read});
        Scheduler.Enqueue(UserProcess1);

        KrUserProcess UserProcess2;
        UserProcess2.Name = "eee";

UserProcess2.IORequests.push_back(KrIORequest{1500,
KrIOOperationType::Read});
        Scheduler.Enqueue(UserProcess2);
    }

    // Own example
    if (ExampleIndex == 0)
    {
        KrUserProcess UserProcess0;
        UserProcess0.Name = "yyy";
```

```
UserProcess0.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Read});

UserProcess0.IORequests.push_back(KrIORequest{110,
KrIOOperationType::Read});

UserProcess0.IORequests.push_back(KrIORequest{500,
KrIOOperationType::Read});

UserProcess0.IORequests.push_back(KrIORequest{510,
KrIOOperationType::Read});

UserProcess0.IORequests.push_back(KrIORequest{1000,
KrIOOperationType::Write});

UserProcess0.IORequests.push_back(KrIORequest{1010,
KrIOOperationType::Write});

UserProcess0.IORequests.push_back(KrIORequest{1500,
KrIOOperationType::Write});

UserProcess0.IORequests.push_back(KrIORequest{1510,
KrIOOperationType::Write});
        Scheduler.Enqueue(UserProcess0);

        KrUserProcess UserProcess1;
        UserProcess1.Name = "qqq";

UserProcess1.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Read});
```

```
UserProcess1.IORequests.push_back(KrIORequest{110,
KrIOOperationType::Read});

UserProcess1.IORequests.push_back(KrIORequest{500,
KrIOOperationType::Write});

UserProcess1.IORequests.push_back(KrIORequest{510,
KrIOOperationType::Write});

UserProcess1.IORequests.push_back(KrIORequest{1000,
KrIOOperationType::Read});

UserProcess1.IORequests.push_back(KrIORequest{1010,
KrIOOperationType::Read});

UserProcess1.IORequests.push_back(KrIORequest{1500,
KrIOOperationType::Write});

UserProcess1.IORequests.push_back(KrIORequest{1510,
KrIOOperationType::Write});
        Scheduler.Enqueue(UserProcess1);

        KrUserProcess UserProcess2;
        UserProcess2.Name = "eee";

UserProcess2.IORequests.push_back(KrIORequest{100,
KrIOOperationType::Read});

UserProcess2.IORequests.push_back(KrIORequest{110,
```

```
KrIOOperationType::Write});

UserProcess2.IORequests.push_back(KrIORequest{500,
KrIOOperationType::Read});

UserProcess2.IORequests.push_back(KrIORequest{510,
KrIOOperationType::Write});

UserProcess2.IORequests.push_back(KrIORequest{1000,
KrIOOperationType::Read});

UserProcess2.IORequests.push_back(KrIORequest{1010,
KrIOOperationType::Write});

UserProcess2.IORequests.push_back(KrIORequest{1500,
KrIOOperationType::Read});

UserProcess2.IORequests.push_back(KrIORequest{1510,
KrIOOperationType::Write});
        Scheduler.Enqueue(UserProcess2);
    }

    std::cout << "\n";

    while (Scheduler.Tick());

    delete Driver;

    return 0;
}
```

Scheduler.h

```
#ifndef KRSPZ_SCHEDULER_H
#define KRSPZ_SCHEDULER_H

#include "UserProcess.h"

#include <vector>

class KrBufferCache;
class KrDriver;

class KrScheduler final
{
public:
    void SetBufferCache(KrBufferCache* const
InBufferCache);
    void SetDriver(KrDriver* const InDriver);

    /* Completes one iteration of the scheduler, returns
whether the next iteration should be called */
    bool Tick();

    void Enqueue(const KrUserProcess& UserProcess);
    KrUserProcess* GetUserProcessByName(const
std::string& UserProcessName);

    /* Wakes up user process with a possible buffer
modification */
    void WakeUp(KrUserProcess& UserProcess);
    /* "Plans" next driver interruption */
```

```
    void RegisterDriverInterruption(const unsigned
TimeUntilDriverInterruption);

    void PrintSettings() const;

private:
    bool UpdateCurrentUserProcess();

    /* Spend time in user process with possible
interruptions */
    void SpendTime(unsigned Time, const std::string&
Mode);
    void SpendTimeInUserProcess(const unsigned Time,
const std::string& Mode);
    void SpendTimeInDriverInterruption();

    unsigned SysCallReadTime = 150;
    unsigned SysCallWriteTime = 150;
    unsigned DriverInterruptionTime = 50;
    unsigned ProcessingAfterReadTime = 7000;
    unsigned ProcessingBeforeWriteTime = 7000;

    unsigned SystemTime = 0;
    unsigned UserProcessTimeAfterDriverInterruption = 0;
    unsigned NextDriverInterruptionSystemTime = 0;

    std::vector<KrUserProcess> UserProcesses;
    size_t CurrentUserProcessIndex = 0;

    KrBufferCache* BufferCache = nullptr;
```

```
    KrDriver* Driver = nullptr;
};


#endif //KRSPZ_SCHEDULER_H
```

BufferCache.h

```
#ifndef KRSPZ_BUFFERCACHE_H
#define KRSPZ_BUFFERCACHE_H

#include "Buffer.h"
#include "IORequest.h"

#include <vector>

class KrDriver;

class KrBufferCache final
{
public:
    void SetDriver(KrDriver* const InDriver);

    /* Request buffer access, returns whether a cache hit
happened */
    bool RequestBuffer(const KrIORequest& IORequest);
    void ModifyBuffer(const unsigned Sector);

    bool Flush();

    /* On buffer read from the disk */
    void OnReadBuffer(const unsigned Sector);
    /* On buffer written to the disk */
    void OnWriteBuffer(const unsigned Sector);

    void PrintBuffer() const;
```

```
    void PrintSettings() const;

private:
    void MarkDirty(const unsigned Sector, const bool
bDirty);

    size_t BufferNum = 7;
    size_t SegmentRightBufferNum = 3;

    std::vector<KrBuffer> SegmentLeft;
    std::vector<KrBuffer> SegmentRight;

    KrDriver* Driver = nullptr;
};

#endif //KRSPZ_BUFFERCACHE_H
```

Driver.h

```
#ifndef KRSPZ_DRIVER_H
#define KRSPZ_DRIVER_H

#include "IORequest.h"

#include <vector>

class KrBufferCache;
class KrScheduler;

class KrDriver
{
public:
    virtual ~KrDriver() = default;

    void SetScheduler(KrScheduler* InScheduler);
    void SetBufferCache(KrBufferCache* const
InBufferCache);

    /* Request disk IO operation */
    void Request(const KrIORequest& IORequest);

    unsigned GetTrackBySector(const unsigned Sector)
const;

    /* On disk driver interruption */
    void OnInterruption();

    virtual void PrintSettings() const;
```

```
protected:
    virtual void AddIORequest(const KrIORequest&
IORequest) = 0;
    virtual void RemoveIORequest(const KrIORequest&
IORequest) = 0;
    virtual std::vector<KrIORequest> GetIORequestQueue()
const = 0;

    virtual void NextIORequest() = 0;

    void SetCurrentIORequest(const KrIORequest* const
InCurrentIORequest);
    const KrIORequest* GetCurrentIORequest() const;

    unsigned GetCurrentTrack() const;

private:
    unsigned TrackNum = 10;
    unsigned SectorPerTrackNum = 500;
    unsigned HeadMoveSingleTrackTime = 500;
    unsigned HeadRewindTime = 10;
    unsigned RotationDelayTime = 4000;
    unsigned SectorAccessTime = 16;

    /* Executes next IO request if not executing yet */
    void ExecuteNextIORequest();

    unsigned CurrentTrack = 0;
```

```
    KrIORequest CurrentIORequest;
    bool bCurrentIORequestSet = false;
    bool bMoveRequested = false;

    KrScheduler* Scheduler = nullptr;
    KrBufferCache* BufferCache = nullptr;
};

#endif //KRSPZ_DRIVER_H
```

DriverFIFO.h

```
#ifndef KRSPZ_DRIVERFIFO_H
#define KRSPZ_DRIVERFIFO_H

#include "Driver.h"

class KrDriverFIFO final : public KrDriver
{
private:
    virtual void AddIORequest(const KrIORequest&
IORequest) override;
    virtual void RemoveIORequest(const KrIORequest&
IORequest) override;
    virtual std::vector<KrIORequest> GetIORequestQueue()
const override;

    virtual void NextIORequest() override;

    std::vector<KrIORequest> IORequestQueue;
};

#endif //KRSPZ_DRIVERFIFO_H
```

DriverLOOK.h

```
#ifndef KRSPZ_DRIVERLOOK_H
#define KRSPZ_DRIVERLOOK_H

#include "Driver.h"

class KrDriverLOOK final : public KrDriver
{
public:
    virtual void PrintSettings() const override;

private:
    virtual void AddIORequest(const KrIORequest&
IORequest) override;
    virtual void RemoveIORequest(const KrIORequest&
IORequest) override;
    virtual std::vector<KrIORequest> GetIORequestQueue()
const override;

    virtual void NextIORequest() override;

    unsigned MaxConsecutiveAccessToTrackNum = 2;

    std::vector<KrIORequest> IORequestQueue;
    bool bMovingOut = true;
    unsigned CurrentConsecutiveAccessToTrackNum = 0;
};

#endif //KRSPZ_DRIVERLOOK_H
```

DriverFLOOK.h

```
#ifndef KRSPZ_DRIVERFLOOK_H
#define KRSPZ_DRIVERFLOOK_H

#include "Driver.h"

class KrDriverFLOOK final : public KrDriver
{
private:
    virtual void AddIORequest(const KrIORequest&
IORequest) override;
    virtual void RemoveIORequest(const KrIORequest&
IORequest) override;
    virtual std::vector<KrIORequest> GetIORequestQueue()
const override;

    virtual void NextIORequest() override;

    std::vector<KrIORequest> IORequestQueueLeft;
    std::vector<KrIORequest> IORequestQueueRight;
    bool bUsingQueueLeft = false;
    bool bMovingOut = true;
};

#endif //KRSPZ_DRIVERFLOOK_H
```

UserProcess.h

```
#ifndef KRSPZ_USERPROCESS_H
#define KRSPZ_USERPROCESS_H

#include "IORequest.h"

#include <string>
#include <vector>

struct KrUserProcess final
{
    std::string Name;
    std::vector<KrIORequest> IORequests;
};

#endif //KRSPZ_USERPROCESS_H
```

IORequest.h

```
#ifndef KRSPZ_IOREQUEST_H
#define KRSPZ_IOREQUEST_H

#include <string>

struct KrUserProcess;

enum class KrIOOperationType
{
    Read,
    Write,
};

enum class KrIORequestState
{
    BeforeIO,
    IOSysCall,
    IOBlocked,
    IONotBlocked,
};

struct KrIORequest final
{
    unsigned Sector;
    KrIOOperationType OperationType;

    /* Whether the sector should be read before writing
*/
    bool bReadFirstly = false;
```

```
    KrIORequestState State = KrIORequestState::BeforeIO;
    std::string UserProcessName;

    bool operator==(const KrIORequest& Other) const
    {
        return Sector == Other.Sector
            && OperationType == Other.OperationType
            && bReadFirstly == Other.bReadFirstly;
    }

    bool operator<(const KrIORequest& Other) const
    {
        return Sector < Other.Sector;
    }
};

#endif //KRSPZ_IOREQUEST_H
```

Buffer.h

```
#ifndef KRSPZ_BUFFER_H
#define KRSPZ_BUFFER_H

struct KrBuffer final
{
    unsigned Sector;
    /* Whether content was modified and not written to
the disk */
    bool bDirty = false;

    bool operator==(const KrBuffer& Other) const
    {
        return Sector == Other.Sector;
    }

    bool operator<(const KrBuffer& Other) const
    {
        return Sector < Other.Sector;
    }
};

#endif //KRSPZ_BUFFER_H
```

Scheduler.cpp

```
#include "Scheduler.h"

#include "BufferCache.h"
#include "Driver.h"

#include <iostream>

void KrScheduler::SetBufferCache(KrBufferCache* const
InBufferCache)
{
    BufferCache = InBufferCache;
}

void KrScheduler::SetDriver(KrDriver* const InDriver)
{
    Driver = InDriver;
}

bool KrScheduler::Tick()
{
    std::cout << "SCHEDULER: " << SystemTime << "us (NEXT
ITERATION)\n";

    // Update current user process
    // and if there are no non-blocked ones
    if (!UpdateCurrentUserProcess())
    {
        // If there are no interruptions "planned"
        if (NextDriverInterruptionSystemTime == 0)
```

```
        {
            std::cout << "SCHEDULER: All user processes
finished, flushing buffer cache\n";

            // Try flushing the buffer cache
            if (BufferCache->Flush())
            {
                std::cout << "\n";
                return true;
            }

            // If the buffer cache is already flushed
            std::cout << "SCHEDULER: Buffer cache
flushed, exiting\n\n";
            return false;
        }

        // If the interruption must happen now
        if (SystemTime ==
NextDriverInterruptionSystemTime)
        {
            SpendTimeInDriverInterruption();
        }
        // Otherwise wait until the next interruption
        else
        {
            const unsigned TimeUntilDriverInterruption =
NextDriverInterruptionSystemTime - SystemTime;
            std::cout << "SCHEDULER: Nothing to do for "
<< TimeUntilDriverInterruption << "us\n\n";
```

```
            SystemTime += TimeUntilDriverInterruption;
        }

        return true;
    }

    KrUserProcess& CurrentUserProcess =
UserProcesses[CurrentUserProcessIndex];
    std::cout << "SCHEDULER: " << "Running user process
\"" << CurrentUserProcess.Name << "\"";

    KrIORequest& IORequest =
CurrentUserProcess.IORequests.front();
    if (IORequest.State == KrIORequestState::BeforeIO)
    {
        std::cout << " in user mode\n";

        if (IORequest.OperationType ==
KrIOOperationType::Write)
        {
            SpendTime(ProcessingBeforeWriteTime, "user");
        }

        std::cout << "SCHEDULER: User process \"" <<
CurrentUserProcess.Name << "\"";
        std::cout << " invoked " <<
(IORequest.OperationType == KrIOOperationType::Read ?
"read" : "write") << "()";
        std::cout << " for buffer (" << Driver-
>GetTrackBySector(IORequest.Sector) << ":" <<
```

```
IORequest.Sector << ")\n";

        IORequest.State = KrIORequestState::IOSysCall;
    }
    else if (IORequest.State ==
KrIORequestState::IOSysCall)
    {
        std::cout << " in kernel mode\n";

        const unsigned TimeSpent =
IORequest.OperationType == KrIOOperationType::Read ?
SysCallReadTime : SysCallWriteTime;
        SpendTime(TimeSpent, "kernel");

        // Request buffer from the cache
        IORequest.UserProcessName =
CurrentUserProcess.Name;
        const bool bCacheHit = BufferCache-
>RequestBuffer(IORequest);
        if (bCacheHit)
        {
            // Do not block user process and execute
buffer modification if necessary
            IORequest.State =
KrIORequestState::IONotBlocked;
            WakeUp(CurrentUserProcess);
        }
        else
        {
            // Block user process
```

```
            IORequest.State =
KrIORequestState::IOBlocked;
            std::cout << "SCHEDULER: Block user process
\"" << CurrentUserProcess.Name << "\"\n";
        }
    }
    else if (IORequest.State ==
KrIORequestState::IONotBlocked)
    {
        std::cout << " in user mode\n";

        if (IORequest.OperationType ==
KrIOOperationType::Read)
        {
            SpendTime(ProcessingAfterReadTime, "user");
        }

        // Remove current IO request from the user
process

CurrentUserProcess.IORequests.erase(CurrentUserProcess.IO
Requests.begin());

        // Finish user process if it has no more IO
requests
        if (CurrentUserProcess.IORequests.empty())
        {
            std::cout << "SCHEDULER: User process \"" <<
CurrentUserProcess.Name << "\" exited\n";
            UserProcesses.erase(UserProcesses.begin() +
```

```
CurrentUserProcessIndex);
            if (CurrentUserProcessIndex ==
UserProcesses.size())
            {
                CurrentUserProcessIndex = 0;
            }
        }
    }

    std::cout << "\n";
    return true;
}

void KrScheduler::Enqueue(const KrUserProcess&
UserProcess)
{
    UserProcesses.push_back(UserProcess);
    std::cout << "SCHEDULER: Enqueue user process \"" <<
UserProcess.Name << "\": [";
    for (const KrIORequest& IORequest :
UserProcess.IORequests)
    {
        std::cout << " ";
        std::cout << (IORequest.OperationType ==
KrIOOperationType::Read ? "R" : "W");
        std::cout << IORequest.Sector;
    }
    std::cout << " ]\n";
}
```

```
KrUserProcess* KrScheduler::GetUserProcessByName(const
std::string& UserProcessName)
{
    for (KrUserProcess& UserProcess : UserProcesses)
    {
        if (UserProcess.Name == UserProcessName)
        {
            return &UserProcess;
        }
    }

    return nullptr;
}

void KrScheduler::WakeUp(KrUserProcess& UserProcess)
{
    KrIORequest& IORequest =
UserProcess.IORequests.front();

    if (IORequest.State == KrIORequestState::IOBlocked)
    {
        IORequest.State = KrIORequestState::IONotBlocked;
        std::cout << "SCHEDULER: Wake up user process \""
<< UserProcess.Name << "\"\n";
    }

    if (IORequest.OperationType ==
KrIOOperationType::Write)
    {
        std::cout << "SCHEDULER: User process \"" <<
```

```
UserProcess.Name << "\"";
        std::cout << " modified buffer (" << Driver-
>GetTrackBySector(IORequest.Sector) << ":" <<
IORequest.Sector << ")\n";

        BufferCache->ModifyBuffer(IORequest.Sector);
    }
}

void KrScheduler::RegisterDriverInterruption(const
unsigned TimeUntilDriverInterruption)
{
    // "Plan" next interruption
    NextDriverInterruptionSystemTime = SystemTime +
UserProcessTimeAfterDriverInterruption +
TimeUntilDriverInterruption;
    std::cout << "SCHEDULER: Next driver interruption at
" << NextDriverInterruptionSystemTime << "us\n";
}

void KrScheduler::PrintSettings() const
{
    std::cout << "\tSysCallReadTime " << SysCallReadTime
<< "\n";
    std::cout << "\tSysCallWriteTime " <<
SysCallWriteTime << "\n";
    std::cout << "\tDriverInterruptionTime " <<
DriverInterruptionTime << "\n";
    std::cout << "\tProcessingAfterReadTime " <<
ProcessingAfterReadTime << "\n";
```

```
    std::cout << "\tProcessingBeforeWriteTime " <<
ProcessingBeforeWriteTime << "\n";
}

bool KrScheduler::UpdateCurrentUserProcess()
{
    if (UserProcesses.empty())
    {
        return false;
    }

    // Find active (that is not blocked by IO operation)
user process
    // beginning from the current one and searching in a
circle
    size_t Index = CurrentUserProcessIndex;
    do
    {
        const std::vector<KrIORequest>& IORequests =
UserProcesses[Index].IORequests;
        if (!IORequests.empty() &&
IORequests.front().State != KrIORequestState::IOBlocked)
        {
            CurrentUserProcessIndex = Index;
            return true;
        }

        ++Index;
        if (Index == UserProcesses.size())
        {
```

```
            Index = 0;
        }
    }
    while (Index != CurrentUserProcessIndex);

    return false;
}

void KrScheduler::SpendTime(unsigned Time, const
std::string& Mode)
{
    // While user process time is not spent
    while (Time > 0)
    {
        UserProcessTimeAfterDriverInterruption = 0;

        // If no interruptions are "planned"
        if (NextDriverInterruptionSystemTime == 0)
        {
            SpendTimeInUserProcess(Time, Mode);
            return;
        }

        // If no interruption can happen during user
process
        const unsigned TimeUntilDriverInterruption =
NextDriverInterruptionSystemTime - SystemTime;
        if (TimeUntilDriverInterruption >= Time)
        {
            SpendTimeInUserProcess(Time, Mode);
```

```
            return;
        }

        // Spend time before the interruption on the user
process

SpendTimeInUserProcess(TimeUntilDriverInterruption,
Mode);
        Time -= TimeUntilDriverInterruption;

        UserProcessTimeAfterDriverInterruption = Time;

        SpendTimeInDriverInterruption();
    }
}

void KrScheduler::SpendTimeInUserProcess(const unsigned
Time, const std::string& Mode)
{
    SystemTime += Time;

    const KrUserProcess& CurrentUserProcess =
UserProcesses[CurrentUserProcessIndex];
    std::cout << "... User process \"" <<
CurrentUserProcess.Name << "\" spent " << Time << "us in
" << Mode << " mode\n";
}

void KrScheduler::SpendTimeInDriverInterruption()
{
```

```
    // Clear the interruption timer
    NextDriverInterruptionSystemTime = 0;

    std::cout << "\n<<< Begin driver interruption at " <<
SystemTime << "us\n";

    SystemTime += DriverInterruptionTime;

    // Call the interruption on the driver
    Driver->OnInterruption();

    std::cout << "... Driver interruption spent " <<
DriverInterruptionTime << "us\n";

    std::cout << ">>> End driver interruption\n\n";
}
```

BufferCache.cpp

```
#include "BufferCache.h"

#include "Driver.h"

#include <iostream>

void KrBufferCache::SetDriver(KrDriver* const InDriver)
{
    Driver = InDriver;
}

bool KrBufferCache::RequestBuffer(const KrIORequest&
IORequest)
{
    std::cout << "CACHE: Requested " <<
(IORequest.OperationType == KrIOOperationType::Read ?
"read" : "write");
    std::cout << " for buffer (" << Driver-
>GetTrackBySector(IORequest.Sector) << ":" <<
IORequest.Sector << ")\n";

    KrBuffer Buffer;
    Buffer.Sector = IORequest.Sector;

    std::vector<KrBuffer>::iterator BufferInSegmentLeft =
std::find(SegmentLeft.begin(), SegmentLeft.end(),
Buffer);
    const bool bCacheHitLeft = BufferInSegmentLeft !=
SegmentLeft.end();
```

```
    const bool bCacheHitRight = !bCacheHitLeft &&
(std::find(SegmentRight.begin(), SegmentRight.end(),
Buffer) != SegmentRight.end());

    const bool bCacheHit = bCacheHitLeft ||
bCacheHitRight;
    std::cout << "CACHE: Buffer (" << Driver-
>GetTrackBySector(IORequest.Sector) << ":" <<
IORequest.Sector << ")";
    std::cout << " " << (bCacheHit ? "found" : "not
found") << "\n";

    if (bCacheHit)
    {
        // If hit buffer is in the left segment
        if (bCacheHitLeft)
        {
            if (SegmentRight.size() <
SegmentRightBufferNum)
            {
                // Move to the right segment
                const KrBuffer& MovedBuffer =
*BufferInSegmentLeft;
                SegmentRight.insert(SegmentRight.begin(),
MovedBuffer);
                SegmentLeft.erase(BufferInSegmentLeft);
            }
            else
            {
                // Move to the beginning of the left
```

```
segment
                const KrBuffer MovedBuffer =
*BufferInSegmentLeft;
                SegmentLeft.erase(BufferInSegmentLeft);
                SegmentLeft.insert(SegmentLeft.begin(),
MovedBuffer);
            }
        }

        PrintBuffer();

        return true;
    }
    else
    {
        KrIORequest IOReadRequest = IORequest;
        IOReadRequest.bReadFirstly =
IOReadRequest.OperationType == KrIOOperationType::Write;

        // If there is some space in the left segment
        if (SegmentLeft.size() < BufferNum -
SegmentRightBufferNum)
        {
            std::cout << "CACHE: Getting free buffer\n";
        }
        else
        {
            const KrBuffer& RemovedBuffer =
SegmentLeft.back();
            std::cout << "CACHE: Removing buffer (" <<
```

```
Driver->GetTrackBySector(RemovedBuffer.Sector) << ":" <<
RemovedBuffer.Sector << ") from cache\n";

            // Request write operation for removed buffer
if its content is dirty
            if (RemovedBuffer.bDirty)
            {
                KrIORequest IOWriteRequest;
                IOWriteRequest.Sector =
RemovedBuffer.Sector;
                IOWriteRequest.OperationType =
KrIOOperationType::Write;
                IOWriteRequest.bReadFirstly = false;
                Driver->Request(IOWriteRequest);
            }

            // Remove the rightmost buffer from the left
segment
            SegmentLeft.erase(SegmentLeft.end() - 1);
        }

        PrintBuffer();

        // Request read operation with a possible buffer
modification after completion
        std::cout << "CACHE: Requesting driver read\n";
        Driver->Request(IOReadRequest);
    }

    return false;
```

```
}

void KrBufferCache::ModifyBuffer(const unsigned Sector)
{
    MarkDirty(Sector, true);
}

bool KrBufferCache::Flush()
{
    std::cout << "CACHE: Flushing buffers\n";

    if (SegmentLeft.empty() && SegmentRight.empty())
    {
        std::cout << "CACHE: Already flushed\n";
        return false;
    }

    KrDriver* const InnerDriver = Driver;
    auto FlushSegment =
[InnerDriver](std::vector<KrBuffer>& Segment)
    {
        for (const KrBuffer& Buffer : Segment)
        {
            // Request write operation for the removed
buffer if its content is dirty
            if (Buffer.bDirty)
            {
                KrIORequest IOWriteRequest;
                IOWriteRequest.Sector = Buffer.Sector;
                IOWriteRequest.OperationType =
```

```
KrIOOperationType::Write;
                IOWriteRequest.bReadFirstly = false;
                InnerDriver->Request(IOWriteRequest);
            }
        }

        Segment.clear();
    };

    FlushSegment(SegmentLeft);
    FlushSegment(SegmentRight);

    std::cout << "CACHE: Successfully flushed\n";
    return true;
}

void KrBufferCache::OnReadBuffer(const unsigned Sector)
{
    KrBuffer Buffer;
    Buffer.Sector = Sector;

    // If the buffer is not in any segment yet
    if (std::find(SegmentLeft.begin(), SegmentLeft.end(),
Buffer) == SegmentLeft.end()
        && std::find(SegmentRight.begin(),
SegmentRight.end(), Buffer) == SegmentRight.end())
    {
        // Insert into the beginning of the left one
        SegmentLeft.insert(SegmentLeft.begin(), Buffer);
        std::cout << "CACHE: Buffer (" << Driver-
```

```
>GetTrackBySector(Buffer.Sector) << ":" << Buffer.Sector
<< ") added to cache\n";
    }

    PrintBuffer();
}

void KrBufferCache::OnWriteBuffer(const unsigned Sector)
{
    MarkDirty(Sector, false);
}

void KrBufferCache::PrintBuffer() const
{
    std::cout << "CACHE: Using LRU strategy\n";
    std::cout << " Left segment: [";
    for (const KrBuffer& Buffer : SegmentLeft)
    {
        std::cout << " (" << Driver-
>GetTrackBySector(Buffer.Sector) << ":" << Buffer.Sector
<< ")";
    }
    std::cout << " ]\n";
    std::cout << " Right segment: [";
    for (const KrBuffer& Buffer : SegmentRight)
    {
        std::cout << " (" << Driver-
>GetTrackBySector(Buffer.Sector) << ":" << Buffer.Sector
<< ")";
    }
```

```
    std::cout << " ]\n";
}

void KrBufferCache::PrintSettings() const
{
    std::cout << "\tBufferNum " << BufferNum << "\n";
    std::cout << "\tSegmentRightBufferNum " <<
SegmentRightBufferNum << "\n";
}

void KrBufferCache::MarkDirty(const unsigned Sector,
const bool bDirty)
{
    auto MarkDirtyInSegment = [Sector,
bDirty](std::vector<KrBuffer>& Segment) -> bool
    {
        for (KrBuffer& Buffer : Segment)
        {
            if (Buffer.Sector == Sector)
            {
                Buffer.bDirty = bDirty;
                return true;
            }
        }
        return false;
    };

    if (!MarkDirtyInSegment(SegmentLeft))
    {
        MarkDirtyInSegment(SegmentRight);
```

```
        }

}
```

Driver.cpp

```
#include "Driver.h"

#include "BufferCache.h"
#include "Scheduler.h"

#include <iostream>

void KrDriver::SetScheduler(KrScheduler* InScheduler)
{
    Scheduler = InScheduler;
}

void KrDriver::SetBufferCache(KrBufferCache* const
InBufferCache)
{
    BufferCache = InBufferCache;
}

void KrDriver::Request(const KrIORequest& IORequest)
{
    // If there are IO operations for the requested
sector
    // no need to read the sector from the disk
    std::vector<KrIORequest> IORequests =
GetIORequestQueue();
    if (const KrIORequest* const InnerCurrentIORequest =
GetCurrentIORequest())
    {
        IORequests.push_back(*InnerCurrentIORequest);
```

```
    }
    for (const KrIORequest& OtherIORequest : IORequests)
    {
        if (OtherIORequest.Sector == IORequest.Sector)
        {
            const unsigned Track =
GetTrackBySector(IORequest.Sector);
            std::cout << "DRIVER: Already requested IO
(read)";
            std::cout << " for buffer (" << Track << ":"
<< IORequest.Sector << ")\n";

            KrIORequest IORequestCopy = IORequest;
            IORequestCopy.bReadFirstly = false;
            AddIORequest(IORequestCopy);

            return;
        }
    }

    AddIORequest(IORequest);

    const bool bRead = IORequest.OperationType ==
KrIOOperationType::Read || IORequest.bReadFirstly;
    const unsigned Track =
GetTrackBySector(IORequest.Sector);
    std::cout << "DRIVER: Requested IO (" << (bRead ?
"read" : "write") << ")";
    std::cout << " for buffer (" << Track << ":" <<
IORequest.Sector << ")\n";
```

```
    ExecuteNextIORequest();
}

unsigned KrDriver::GetTrackBySector(const unsigned
Sector) const
{
    return Sector / SectorPerTrackNum;
}

void KrDriver::OnInterruption()
{
    bMoveRequested = false;

    const KrIORequest IORequest = *GetCurrentIORequest();
// getting by value cause a reference can become invalid

    const unsigned Track =
GetTrackBySector(IORequest.Sector);
    CurrentTrack = Track;

    const bool bWasRead = IORequest.OperationType ==
KrIOOperationType::Read || IORequest.bReadFirstly;
    std::cout << "DRIVER: Completed IO (" << (bWasRead ?
"read" : "write") << ")";
    std::cout << " for buffer (" << Track << ":" <<
IORequest.Sector << ")\n";

    // Notifying the buffer cache about the IO operation
completion
```

```
    if (bWasRead)
    {
        BufferCache->OnReadBuffer(IORequest.Sector);
    }
    else
    {
        BufferCache->OnWriteBuffer(IORequest.Sector);
    }

    // Waking up the user process if it is still running
    if (KrUserProcess* UserProcess = Scheduler-
>GetUserProcessByName(IORequest.UserProcessName))
    {
        Scheduler->WakeUp(*UserProcess);
    }

    // Waking up other user processes that have requested
IO operation for the same sector
    for (const KrIORequest& OtherIORequest :
GetIORequestQueue())
    {
        if (OtherIORequest.Sector == IORequest.Sector)
        {
            if (KrUserProcess* OtherUserProcess =
Scheduler-
>GetUserProcessByName(OtherIORequest.UserProcessName))
            {
                Scheduler->WakeUp(*OtherUserProcess);
                RemoveIORequest(OtherIORequest);
            }
```

```
        }
    }

    ExecuteNextIORequest();
}

void KrDriver::PrintSettings() const
{
    std::cout << "\tTrackNum " << TrackNum << "\n";
    std::cout << "\tSectorPerTrackNum " <<
SectorPerTrackNum << "\n";
    std::cout << "\tHeadMoveSingleTrackTime " <<
HeadMoveSingleTrackTime << "\n";
    std::cout << "\tHeadRewindTime " << HeadRewindTime <<
"\n";
    std::cout << "\tRotationDelayTime " <<
RotationDelayTime << "\n";
    std::cout << "\tSectorAccessTime " <<
SectorAccessTime << "\n";
}

void KrDriver::SetCurrentIORequest(const KrIORequest*
const InCurrentIORequest)
{
    bCurrentIORequestSet = InCurrentIORequest;
    if (InCurrentIORequest)
    {
        CurrentIORequest = *InCurrentIORequest;
    }
}
```

```
const KrIORequest* KrDriver::GetCurrentIORequest() const
{
    if (bCurrentIORequestSet)
    {
        return &CurrentIORequest;
    }
    return nullptr;
}

unsigned KrDriver::GetCurrentTrack() const
{
    return CurrentTrack;
}

void KrDriver::ExecuteNextIORequest()
{
    if (bMoveRequested)
    {
        return;
    }

    // Get next IO request
    NextIORequest();
    const KrIORequest* const IORequest =
GetCurrentIORequest();

    // If there are no IO requests
    if (!IORequest)
    {
```

```
            std::cout << "DRIVER: Nothing to do\n";
            return;
        }

        bMoveRequested = true;

        // Requested track
        const unsigned Track = GetTrackBySector(IORequest->Sector);

        // Direct move
        const unsigned CurrentDeltaTrack = std::abs((int)Track - (int)CurrentTrack);
        const unsigned HeadMoveDirectTime = CurrentDeltaTrack * HeadMoveSingleTrackTime;

        // Move with rewind
        const unsigned EdgeTrack = (CurrentTrack > TrackNum / 2) ? TrackNum : 0;
        const unsigned EdgeDeltaTrack = std::abs((int)Track - (int)EdgeTrack);
        const unsigned HeadMoveWithRewindTime = (CurrentDeltaTrack ? HeadRewindTime : 0) + EdgeDeltaTrack * HeadMoveSingleTrackTime;

        // Smallest move time
        const unsigned SmallestHeadMoveTime = std::min(HeadMoveDirectTime, HeadMoveWithRewindTime);

        std::cout << "DRIVER: Moving from track " <<
```

```
CurrentTrack << " to " << Track << " in " <<
SmallestHeadMoveTime << "us\n";
    std::cout << " Direct move time: " <<
HeadMoveDirectTime << "us\n";
    std::cout << " Move time with rewind: " <<
HeadMoveWithRewindTime << "us\n";

    const unsigned IOOperationTime = SmallestHeadMoveTime
+ RotationDelayTime + SectorAccessTime;
    std::cout << "DRIVER: Sector access in " <<
IOOperationTime << "us\n";

    // "Plan" an interruption
    Scheduler-
>RegisterDriverInterruption(IOOperationTime);
}
```

DriverFIFO.cpp

```
#include "DriverFIFO.h"

#include <iostream>

void KrDriverFIFO::AddIORequest(const KrIORequest&
IORequest)
{
    // Add new IO request to the end of the queue
    IORequestQueue.push_back(IORequest);
}

void KrDriverFIFO::RemoveIORequest(const KrIORequest&
IORequest)
{

IORequestQueue.erase(std::remove(IORequestQueue.begin(),
IORequestQueue.end(), IORequest), IORequestQueue.end());
}

std::vector<KrIORequest>
KrDriverFIFO::GetIORequestQueue() const
{
    return IORequestQueue;
}

void KrDriverFIFO::NextIORequest()
{
    if (IORequestQueue.empty())
    {
```

```
        SetCurrentIORequest(nullptr);
        return;
    }

    // Get next IO request from the beginning of the
queue
    const KrIORequest Result = IORequestQueue.front();
    IORequestQueue.erase(IORequestQueue.begin());

    std::cout << "DRIVER: Using FIFO strategy\n";
    std::cout << " Current buffer: (" <<
GetTrackBySector(Result.Sector) << ":" << Result.Sector
<< ")\n";
    std::cout << " Buffer queue: [";
    for (const KrIORequest& IORequest : IORequestQueue)
    {
        std::cout << " (" <<
GetTrackBySector(IORequest.Sector) << ":" <<
IORequest.Sector << ")";
    }
    std::cout << " ]\n";

    SetCurrentIORequest(&Result);
}
```

DriverLOOK.cpp

```
#include "DriverLOOK.h"

#include <algorithm>
#include <iostream>

void KrDriverLOOK::PrintSettings() const
{
    KrDriver::PrintSettings();

    std::cout << "\tMaxConsecutiveAccessToTrackNum " <<
MaxConsecutiveAccessToTrackNum << "\n";
}

void KrDriverLOOK::AddIORequest(const KrIORequest&
IORequest)
{
    // And request to the queue
    IORequestQueue.push_back(IORequest);
    // And sort the queue
    std::sort(IORequestQueue.begin(),
IORequestQueue.end());
}

void KrDriverLOOK::RemoveIORequest(const KrIORequest&
IORequest)
{

IORequestQueue.erase(std::remove(IORequestQueue.begin(),
IORequestQueue.end(), IORequest), IORequestQueue.end());
```

```
}

std::vector<KrIORequest>
KrDriverLOOK::GetIORequestQueue() const
{
    return IORequestQueue;
}

void KrDriverLOOK::NextIORequest()
{
    if (IORequestQueue.empty())
    {
        SetCurrentIORequest(nullptr);
        return;
    }

    KrIORequest Result;
    bool bResultIsSet = false;

    // Determine start and end indexes for the loop based
on the move direction
    size_t Index, End;
    if (bMovingOut)
    {
        Index = 0;
        End = IORequestQueue.size();
    }
    else
    {
        Index = IORequestQueue.size() - 1;
```

```
        End = -1;
    }
    while (Index != End)
    {
        const KrIORequest& IORequest =
IORequestQueue[Index];
        const unsigned Track =
GetTrackBySector(IORequest.Sector);

        // If requested track is the current one
        if (Track == GetCurrentTrack())
        {
            // If able to access the same track one more
time
            if (CurrentConsecutiveAccessToTrackNum <
MaxConsecutiveAccessToTrackNum)
            {
                ++CurrentConsecutiveAccessToTrackNum;

                Result = IORequest;
                bResultIsSet = true;

IORequestQueue.erase(IORequestQueue.begin() + Index);

                break;
            }
        }
        // If requested track lays in the move direction
        else if ((Track > GetCurrentTrack()) ==
bMovingOut)
```

```
        {
            // Clear the consecutive access to track
counter
            CurrentConsecutiveAccessToTrackNum = 0;

            Result = IORequest;
            bResultIsSet = true;
            IORequestQueue.erase(IORequestQueue.begin() +
Index);

            break;
        }

        if (bMovingOut)
        {
            ++Index;
        }
        else
        {
            --Index;
        }
    }

    // If unable move in the current direction
    if (!bResultIsSet)
    {
        // Flip the move direction, clear the consecutive
access to track counter and try one more time
        bMovingOut = !bMovingOut;
        CurrentConsecutiveAccessToTrackNum = 0;
```

```
        NextIORequest();
    }
    else
    {
        SetCurrentIORequest(&Result);

        std::cout << "DRIVER: Using LOOK strategy\n";
        std::cout << " Direction: " << (bMovingOut ?
"OUT" : "IN") << "\n";
        std::cout << " Current buffer: (" <<
GetTrackBySector(GetCurrentIORequest()->Sector) << ":" <<
GetCurrentIORequest()->Sector << ")\n";
        std::cout << " Buffer queue: [";
        for (const KrIORequest& IORequest :
IORequestQueue)
        {
            std::cout << " (" <<
GetTrackBySector(IORequest.Sector) << ":" <<
IORequest.Sector << ")";
        }
        std::cout << " ]\n";
    }
}
```

DriverFLOOK.cpp

```cpp
#include "DriverFLOOK.h"

#include <algorithm>
#include <iostream>

void KrDriverFLOOK::AddIORequest(const KrIORequest&
IORequest)
{
    // Get the inactive queue and add the request to it
    std::vector<KrIORequest>& IORequestQueue =
bUsingQueueLeft ? IORequestQueueRight :
IORequestQueueLeft;
    IORequestQueue.push_back(IORequest);
    // Then sort the queue
    std::sort(IORequestQueue.begin(),
IORequestQueue.end());
}

void KrDriverFLOOK::RemoveIORequest(const KrIORequest&
IORequest)
{
    std::vector<KrIORequest>::iterator Iter =
std::remove(IORequestQueueLeft.begin(),
IORequestQueueLeft.end(), IORequest);
    if (Iter != IORequestQueueLeft.end())
    {
        IORequestQueueLeft.erase(Iter,
IORequestQueueLeft.end());
    }
```

```
    else
    {
        Iter = std::remove(IORequestQueueRight.begin(),
IORequestQueueRight.end(), IORequest);
        IORequestQueueRight.erase(Iter,
IORequestQueueRight.end());
    }
}

std::vector<KrIORequest>
KrDriverFLOOK::GetIORequestQueue() const
{
    std::vector<KrIORequest> Result;
    Result.reserve(IORequestQueueLeft.size() +
IORequestQueueRight.size());
    Result.insert(Result.begin(),
IORequestQueueLeft.begin(), IORequestQueueLeft.end());
    Result.insert(Result.begin(),
IORequestQueueRight.begin(), IORequestQueueRight.end());
    return Result;
}

void KrDriverFLOOK::NextIORequest()
{
    if (IORequestQueueLeft.empty() &&
IORequestQueueRight.empty())
    {
        SetCurrentIORequest(nullptr);
        return;
    }
```

```
    // If the active is empty
    if (bUsingQueueLeft && IORequestQueueLeft.empty() ||
!bUsingQueueLeft && IORequestQueueRight.empty())
    {
        // Swap queues
        bUsingQueueLeft = !bUsingQueueLeft;
    }
    // And get the active one
    std::vector<KrIORequest>& IORequestQueue =
bUsingQueueLeft ? IORequestQueueLeft :
IORequestQueueRight;

    KrIORequest Result;
    bool bResultIsSet = false;

    // Determine start and end indexes for the loop based
on the move direction
    size_t Index, End;
    if (bMovingOut)
    {
        Index = 0;
        End = IORequestQueue.size();
    }
    else
    {
        Index = IORequestQueue.size() - 1;
        End = -1;
    }
    while (Index != End)
```

```
    {
        const KrIORequest& IORequest =
IORequestQueue[Index];
        const unsigned Track =
GetTrackBySector(IORequest.Sector);

        // If requested track is the current one
        if (Track == GetCurrentTrack())
        {
            Result = IORequest;
            bResultIsSet = true;
            IORequestQueue.erase(IORequestQueue.begin() +
Index);

            break;
        }
        // If requested track lays in the move direction
        else if ((Track > GetCurrentTrack()) ==
bMovingOut)
        {
            Result = IORequest;
            bResultIsSet = true;
            IORequestQueue.erase(IORequestQueue.begin() +
Index);

            break;
        }

        if (bMovingOut)
        {
```

```
            ++Index;
        }
        else
        {
            --Index;
        }
    }

    // If unable move in the current direction
    if (!bResultIsSet)
    {
        // Flip the move direction and try one more time
        bMovingOut = !bMovingOut;
        NextIORequest();
    }
    else
    {
        SetCurrentIORequest(&Result);

        std::cout << "DRIVER: Using FLOOK strategy\n";
        std::cout << " Direction: " << (bMovingOut ?
"OUT" : "IN") << "\n";
        std::cout << " Current buffer: (" <<
GetTrackBySector(GetCurrentIORequest()->Sector) << ":" <<
GetCurrentIORequest()->Sector << ")\n";
        std::cout << " Current queue: " <<
(bUsingQueueLeft ? "LEFT" : "RIGHT") << "\n";
        std::cout << " Left buffer queue: [";
        for (const KrIORequest& IORequest :
IORequestQueueLeft)
```

```
        {
            std::cout << " (" <<
GetTrackBySector(IORequest.Sector) << ":" <<
IORequest.Sector << ")";
        }
        std::cout << " ]\n";
        std::cout << " Right buffer queue: [";
        for (const KrIORequest& IORequest :
IORequestQueueRight)
        {
            std::cout << " (" <<
GetTrackBySector(IORequest.Sector) << ":" <<
IORequest.Sector << ")";
        }
        std::cout << " ]\n";
    }
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.26)
project(krspz)

set(CMAKE_CXX_STANDARD 20)

add_executable(krspz main.cpp
        Scheduler.h
        UserProcess.h
        IORequest.h
        BufferCache.h
        Driver.h
        Scheduler.cpp
        Driver.cpp
        BufferCache.cpp
        Buffer.h
        DriverFIFO.h
        DriverFIFO.cpp
        DriverLOOK.h
        DriverLOOK.cpp
        DriverFLOOK.h
        DriverFLOOK.cpp
)
```

Приклади виводу програми про дії в системі під час моделювання

Приклад №1

```
Enter example index (0-8):4
Enter disk driver strategy (1-3):1

CONSOLE: Change MaxConsecutiveAccessToTrackNum in the
header file

CONSOLE: Change BufferNum and SegmentRightBufferNum in
the header file

Settings:
        SysCallReadTime 150
        SysCallWriteTime 150
        DriverInterruptionTime 50
        ProcessingAfterReadTime 7000
        ProcessingBeforeWriteTime 7000
        BufferNum 7
        SegmentRightBufferNum 3
        TrackNum 10
        SectorPerTrackNum 500
        HeadMoveSingleTrackTime 500
        HeadRewindTime 10
        RotationDelayTime 4000
        SectorAccessTime 16

SCHEDULER: Enqueue user process "yyy": [ R100 ]
SCHEDULER: Enqueue user process "qqq": [ R100 ]

SCHEDULER: 0us (NEXT ITERATION)
```

```
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" invoked read() for buffer
(0:100)

SCHEDULER: 0us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested read for buffer (0:100)
CACHE: Buffer (0:100) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:100)
DRIVER: Using FIFO strategy
 Current buffer: (0:100)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 4166us
SCHEDULER: Block user process "yyy"

SCHEDULER: 150us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in user mode
SCHEDULER: User process "qqq" invoked read() for buffer
(0:100)
```

```
SCHEDULER: 150us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in kernel mode
... User process "qqq" spent 150us in kernel mode
CACHE: Requested read for buffer (0:100)
CACHE: Buffer (0:100) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Already requested IO (read) for buffer (0:100)
SCHEDULER: Block user process "qqq"

SCHEDULER: 300us (NEXT ITERATION)
SCHEDULER: Nothing to do for 3866us

SCHEDULER: 4166us (NEXT ITERATION)

<<< Begin driver interruption at 4166us
DRIVER: Completed IO (read) for buffer (0:100)
CACHE: Buffer (0:100) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
SCHEDULER: Wake up user process "qqq"
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption
```

```
SCHEDULER: 4216us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in user mode
... User process "qqq" spent 7000us in user mode
SCHEDULER: User process "qqq" exited

SCHEDULER: 11216us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
... User process "yyy" spent 7000us in user mode
SCHEDULER: User process "yyy" exited

SCHEDULER: 18216us (NEXT ITERATION)
SCHEDULER: All user processes finished, flushing buffer
cache
CACHE: Flushing buffers
CACHE: Successfully flushed

SCHEDULER: 18216us (NEXT ITERATION)
SCHEDULER: All user processes finished, flushing buffer
cache
CACHE: Flushing buffers
CACHE: Already flushed
SCHEDULER: Buffer cache flushed, exiting
```

Приклад №2

```
Enter example index (0-8):5
Enter disk driver strategy (1-3):1

CONSOLE: Change MaxConsecutiveAccessToTrackNum in the
header file

CONSOLE: Change BufferNum and SegmentRightBufferNum in
the header file

Settings:
        SysCallReadTime 150
        SysCallWriteTime 150
        DriverInterruptionTime 50
        ProcessingAfterReadTime 7000
        ProcessingBeforeWriteTime 7000
        BufferNum 7
        SegmentRightBufferNum 0
        TrackNum 10
        SectorPerTrackNum 500
        HeadMoveSingleTrackTime 500
        HeadRewindTime 10
        RotationDelayTime 4000
        SectorAccessTime 16

SCHEDULER: Enqueue user process "yyy": [ W100 R110 R120
R130 R140 R150 R160 W170 ]

SCHEDULER: 0us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
```

```
... User process "yyy" spent 7000us in user mode
SCHEDULER: User process "yyy" invoked write() for buffer
(0:100)

SCHEDULER: 7000us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested write for buffer (0:100)
CACHE: Buffer (0:100) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:100)
DRIVER: Using FIFO strategy
 Current buffer: (0:100)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 11166us
SCHEDULER: Block user process "yyy"

SCHEDULER: 7150us (NEXT ITERATION)
SCHEDULER: Nothing to do for 4016us

SCHEDULER: 11166us (NEXT ITERATION)
```

```
<<< Begin driver interruption at 11166us
DRIVER: Completed IO (read) for buffer (0:100)
CACHE: Buffer (0:100) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
SCHEDULER: User process "yyy" modified buffer (0:100)
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 11216us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode

SCHEDULER: 11216us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" invoked read() for buffer
(0:110)

SCHEDULER: 11216us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested read for buffer (0:110)
CACHE: Buffer (0:110) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ (0:100) ]
 Right segment: [ ]
CACHE: Requesting driver read
```

```
DRIVER: Requested IO (read) for buffer (0:110)
DRIVER: Using FIFO strategy
 Current buffer: (0:110)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 15382us
SCHEDULER: Block user process "yyy"

SCHEDULER: 11366us (NEXT ITERATION)
SCHEDULER: Nothing to do for 4016us

SCHEDULER: 15382us (NEXT ITERATION)

<<< Begin driver interruption at 15382us
DRIVER: Completed IO (read) for buffer (0:110)
CACHE: Buffer (0:110) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:110) (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 15432us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
... User process "yyy" spent 7000us in user mode
```

```
SCHEDULER: 22432us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" invoked read() for buffer
(0:120)

SCHEDULER: 22432us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested read for buffer (0:120)
CACHE: Buffer (0:120) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ (0:110) (0:100) ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:120)
DRIVER: Using FIFO strategy
 Current buffer: (0:120)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 26598us
SCHEDULER: Block user process "yyy"

SCHEDULER: 22582us (NEXT ITERATION)
SCHEDULER: Nothing to do for 4016us
```

```
SCHEDULER: 26598us (NEXT ITERATION)

<<< Begin driver interruption at 26598us
DRIVER: Completed IO (read) for buffer (0:120)
CACHE: Buffer (0:120) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:120) (0:110) (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 26648us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
... User process "yyy" spent 7000us in user mode

SCHEDULER: 33648us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" invoked read() for buffer
(0:130)

SCHEDULER: 33648us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested read for buffer (0:130)
CACHE: Buffer (0:130) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ (0:120) (0:110) (0:100) ]
```

```
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:130)
DRIVER: Using FIFO strategy
 Current buffer: (0:130)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 37814us
SCHEDULER: Block user process "yyy"

SCHEDULER: 33798us (NEXT ITERATION)
SCHEDULER: Nothing to do for 4016us

SCHEDULER: 37814us (NEXT ITERATION)

<<< Begin driver interruption at 37814us
DRIVER: Completed IO (read) for buffer (0:130)
CACHE: Buffer (0:130) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:130) (0:120) (0:110) (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 37864us (NEXT ITERATION)
```

```
SCHEDULER: Running user process "yyy" in user mode
... User process "yyy" spent 7000us in user mode

SCHEDULER: 44864us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" invoked read() for buffer
(0:140)

SCHEDULER: 44864us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested read for buffer (0:140)
CACHE: Buffer (0:140) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ (0:130) (0:120) (0:110) (0:100) ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:140)
DRIVER: Using FIFO strategy
 Current buffer: (0:140)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 49030us
SCHEDULER: Block user process "yyy"

SCHEDULER: 45014us (NEXT ITERATION)
```

```
SCHEDULER: Nothing to do for 4016us

SCHEDULER: 49030us (NEXT ITERATION)

<<< Begin driver interruption at 49030us
DRIVER: Completed IO (read) for buffer (0:140)
CACHE: Buffer (0:140) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:140) (0:130) (0:120) (0:110) (0:100)
]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 49080us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
... User process "yyy" spent 7000us in user mode

SCHEDULER: 56080us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" invoked read() for buffer
(0:150)

SCHEDULER: 56080us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested read for buffer (0:150)
CACHE: Buffer (0:150) not found
```

```
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ (0:140) (0:130) (0:120) (0:110) (0:100)
]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:150)
DRIVER: Using FIFO strategy
 Current buffer: (0:150)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 60246us
SCHEDULER: Block user process "yyy"

SCHEDULER: 56230us (NEXT ITERATION)
SCHEDULER: Nothing to do for 4016us

SCHEDULER: 60246us (NEXT ITERATION)

<<< Begin driver interruption at 60246us
DRIVER: Completed IO (read) for buffer (0:150)
CACHE: Buffer (0:150) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:150) (0:140) (0:130) (0:120) (0:110)
(0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
```

```
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 60296us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
... User process "yyy" spent 7000us in user mode

SCHEDULER: 67296us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" invoked read() for buffer
(0:160)

SCHEDULER: 67296us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested read for buffer (0:160)
CACHE: Buffer (0:160) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ (0:150) (0:140) (0:130) (0:120) (0:110)
(0:100) ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:160)
DRIVER: Using FIFO strategy
 Current buffer: (0:160)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
```

```
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 71462us
SCHEDULER: Block user process "yyy"

SCHEDULER: 67446us (NEXT ITERATION)
SCHEDULER: Nothing to do for 4016us

SCHEDULER: 71462us (NEXT ITERATION)

<<< Begin driver interruption at 71462us
DRIVER: Completed IO (read) for buffer (0:160)
CACHE: Buffer (0:160) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:160) (0:150) (0:140) (0:130) (0:120)
(0:110) (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 71512us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
... User process "yyy" spent 7000us in user mode

SCHEDULER: 78512us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
... User process "yyy" spent 7000us in user mode
```

```
SCHEDULER: User process "yyy" invoked write() for buffer
(0:170)

SCHEDULER: 85512us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested write for buffer (0:170)
CACHE: Buffer (0:170) not found
CACHE: Removing buffer (0:100) from cache
DRIVER: Requested IO (write) for buffer (0:100)
DRIVER: Using FIFO strategy
 Current buffer: (0:100)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 89678us
CACHE: Using LRU strategy
 Left segment: [ (0:160) (0:150) (0:140) (0:130) (0:120)
(0:110) ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:170)
SCHEDULER: Block user process "yyy"

SCHEDULER: 85662us (NEXT ITERATION)
SCHEDULER: Nothing to do for 4016us

SCHEDULER: 89678us (NEXT ITERATION)
```

```
<<< Begin driver interruption at 89678us
DRIVER: Completed IO (write) for buffer (0:100)
DRIVER: Using FIFO strategy
 Current buffer: (0:170)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 93744us
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 89728us (NEXT ITERATION)
SCHEDULER: Nothing to do for 4016us

SCHEDULER: 93744us (NEXT ITERATION)

<<< Begin driver interruption at 93744us
DRIVER: Completed IO (read) for buffer (0:170)
CACHE: Buffer (0:170) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:170) (0:160) (0:150) (0:140) (0:130)
(0:120) (0:110) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
SCHEDULER: User process "yyy" modified buffer (0:170)
DRIVER: Nothing to do
... Driver interruption spent 50us
```

```
>>> End driver interruption

SCHEDULER: 93794us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" exited

SCHEDULER: 93794us (NEXT ITERATION)
SCHEDULER: All user processes finished, flushing buffer
cache
CACHE: Flushing buffers
DRIVER: Requested IO (write) for buffer (0:170)
DRIVER: Using FIFO strategy
 Current buffer: (0:170)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 97810us
CACHE: Successfully flushed

SCHEDULER: 93794us (NEXT ITERATION)
SCHEDULER: Nothing to do for 4016us

SCHEDULER: 97810us (NEXT ITERATION)

<<< Begin driver interruption at 97810us
DRIVER: Completed IO (write) for buffer (0:170)
DRIVER: Nothing to do
... Driver interruption spent 50us
```

```
>>> End driver interruption


SCHEDULER: 97860us (NEXT ITERATION)
SCHEDULER: All user processes finished, flushing buffer
cache
CACHE: Flushing buffers
CACHE: Already flushed
SCHEDULER: Buffer cache flushed, exiting
```

Приклад №3

```
Enter example index (0-8):6
Enter disk driver strategy (1-3):1

CONSOLE: Change MaxConsecutiveAccessToTrackNum in the
header file

CONSOLE: Change BufferNum and SegmentRightBufferNum in
the header file

Settings:
        SysCallReadTime 150
        SysCallWriteTime 150
        DriverInterruptionTime 50
        ProcessingAfterReadTime 7000
        ProcessingBeforeWriteTime 7000
        BufferNum 7
        SegmentRightBufferNum 0
        TrackNum 10
        SectorPerTrackNum 500
        HeadMoveSingleTrackTime 500
        HeadRewindTime 10
        RotationDelayTime 4000
        SectorAccessTime 16

SCHEDULER: Enqueue user process "yyy": [ W100 ]
SCHEDULER: Enqueue user process "qqq": [ R100 ]

SCHEDULER: 0us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
```

```
... User process "yyy" spent 7000us in user mode
SCHEDULER: User process "yyy" invoked write() for buffer
(0:100)

SCHEDULER: 7000us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested write for buffer (0:100)
CACHE: Buffer (0:100) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:100)
DRIVER: Using FIFO strategy
 Current buffer: (0:100)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 11166us
SCHEDULER: Block user process "yyy"

SCHEDULER: 7150us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in user mode
SCHEDULER: User process "qqq" invoked read() for buffer
(0:100)
```

```
SCHEDULER: 7150us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in kernel mode
... User process "qqq" spent 150us in kernel mode
CACHE: Requested read for buffer (0:100)
CACHE: Buffer (0:100) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Already requested IO (read) for buffer (0:100)
SCHEDULER: Block user process "qqq"

SCHEDULER: 7300us (NEXT ITERATION)
SCHEDULER: Nothing to do for 3866us

SCHEDULER: 11166us (NEXT ITERATION)

<<< Begin driver interruption at 11166us
DRIVER: Completed IO (read) for buffer (0:100)
CACHE: Buffer (0:100) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
SCHEDULER: User process "yyy" modified buffer (0:100)
SCHEDULER: Wake up user process "qqq"
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption
```

```
SCHEDULER: 11216us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in user mode
... User process "qqq" spent 7000us in user mode
SCHEDULER: User process "qqq" exited

SCHEDULER: 18216us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" exited

SCHEDULER: 18216us (NEXT ITERATION)
SCHEDULER: All user processes finished, flushing buffer
cache
CACHE: Flushing buffers
DRIVER: Requested IO (write) for buffer (0:100)
DRIVER: Using FIFO strategy
 Current buffer: (0:100)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 22232us
CACHE: Successfully flushed

SCHEDULER: 18216us (NEXT ITERATION)
SCHEDULER: Nothing to do for 4016us

SCHEDULER: 22232us (NEXT ITERATION)
```

```
<<< Begin driver interruption at 22232us
DRIVER: Completed IO (write) for buffer (0:100)
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 22282us (NEXT ITERATION)
SCHEDULER: All user processes finished, flushing buffer
cache
CACHE: Flushing buffers
CACHE: Already flushed
SCHEDULER: Buffer cache flushed, exiting
```

Приклад №4

```
Enter example index (0-8):8
Enter disk driver strategy (1-3):2

CONSOLE: Change MaxConsecutiveAccessToTrackNum in the
header file

CONSOLE: Change BufferNum and SegmentRightBufferNum in
the header file

Settings:
        SysCallReadTime 150
        SysCallWriteTime 150
        DriverInterruptionTime 50
        ProcessingAfterReadTime 7000
        ProcessingBeforeWriteTime 7000
        BufferNum 7
        SegmentRightBufferNum 0
        TrackNum 10
        SectorPerTrackNum 500
        HeadMoveSingleTrackTime 500
        HeadRewindTime 10
        RotationDelayTime 4000
        SectorAccessTime 16
        MaxConsecutiveAccessToTrackNum 2

SCHEDULER: Enqueue user process "yyy": [ R100 ]
SCHEDULER: Enqueue user process "qqq": [ R110 ]
SCHEDULER: Enqueue user process "eee": [ R1500 ]
```

```
SCHEDULER: 0us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" invoked read() for buffer
(0:100)

SCHEDULER: 0us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested read for buffer (0:100)
CACHE: Buffer (0:100) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:100)
DRIVER: Using LOOK strategy
 Direction: OUT
 Current buffer: (0:100)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 4166us
SCHEDULER: Block user process "yyy"

SCHEDULER: 150us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in user mode
```

```
SCHEDULER: User process "qqq" invoked read() for buffer
(0:110)

SCHEDULER: 150us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in kernel mode
... User process "qqq" spent 150us in kernel mode
CACHE: Requested read for buffer (0:110)
CACHE: Buffer (0:110) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:110)
SCHEDULER: Block user process "qqq"

SCHEDULER: 300us (NEXT ITERATION)
SCHEDULER: Running user process "eee" in user mode
SCHEDULER: User process "eee" invoked read() for buffer
(3:1500)

SCHEDULER: 300us (NEXT ITERATION)
SCHEDULER: Running user process "eee" in kernel mode
... User process "eee" spent 150us in kernel mode
CACHE: Requested read for buffer (3:1500)
CACHE: Buffer (3:1500) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
```

```
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (3:1500)
SCHEDULER: Block user process "eee"

SCHEDULER: 450us (NEXT ITERATION)
SCHEDULER: Nothing to do for 3716us

SCHEDULER: 4166us (NEXT ITERATION)

<<< Begin driver interruption at 4166us
DRIVER: Completed IO (read) for buffer (0:100)
CACHE: Buffer (0:100) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
DRIVER: Using LOOK strategy
 Direction: OUT
 Current buffer: (0:110)
 Buffer queue: [ (3:1500) ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 8232us
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 4216us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
```

```
... User process "yyy" spent 4016us in user mode

<<< Begin driver interruption at 8232us
DRIVER: Completed IO (read) for buffer (0:110)
CACHE: Buffer (0:110) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:110) (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "qqq"
DRIVER: Using LOOK strategy
 Direction: OUT
 Current buffer: (3:1500)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 3 in 1500us
 Direct move time: 1500us
 Move time with rewind: 1510us
DRIVER: Sector access in 5516us
SCHEDULER: Next driver interruption at 16782us
... Driver interruption spent 50us
>>> End driver interruption

... User process "yyy" spent 2984us in user mode
SCHEDULER: User process "yyy" exited

SCHEDULER: 11266us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in user mode
... User process "qqq" spent 5516us in user mode

<<< Begin driver interruption at 16782us
DRIVER: Completed IO (read) for buffer (3:1500)
```

```
CACHE: Buffer (3:1500) added to cache
CACHE: Using LRU strategy
 Left segment: [ (3:1500) (0:110) (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "eee"
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption

... User process "qqq" spent 1484us in user mode
SCHEDULER: User process "qqq" exited

SCHEDULER: 18316us (NEXT ITERATION)
SCHEDULER: Running user process "eee" in user mode
... User process "eee" spent 7000us in user mode
SCHEDULER: User process "eee" exited

SCHEDULER: 25316us (NEXT ITERATION)
SCHEDULER: All user processes finished, flushing buffer
cache
CACHE: Flushing buffers
CACHE: Successfully flushed

SCHEDULER: 25316us (NEXT ITERATION)
SCHEDULER: All user processes finished, flushing buffer
cache
CACHE: Flushing buffers
CACHE: Already flushed
SCHEDULER: Buffer cache flushed, exiting
```

Приклад №5

```
Enter example index (0-8):7
Enter disk driver strategy (1-3):2

CONSOLE: Change MaxConsecutiveAccessToTrackNum in the
header file

CONSOLE: Change BufferNum and SegmentRightBufferNum in
the header file

Settings:
        SysCallReadTime 150
        SysCallWriteTime 150
        DriverInterruptionTime 50
        ProcessingAfterReadTime 7000
        ProcessingBeforeWriteTime 7000
        BufferNum 7
        SegmentRightBufferNum 0
        TrackNum 10
        SectorPerTrackNum 500
        HeadMoveSingleTrackTime 500
        HeadRewindTime 10
        RotationDelayTime 4000
        SectorAccessTime 16
        MaxConsecutiveAccessToTrackNum 1

SCHEDULER: Enqueue user process "yyy": [ R100 ]
SCHEDULER: Enqueue user process "qqq": [ R110 ]
SCHEDULER: Enqueue user process "eee": [ R1500 ]
```

```
SCHEDULER: 0us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
SCHEDULER: User process "yyy" invoked read() for buffer
(0:100)

SCHEDULER: 0us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in kernel mode
... User process "yyy" spent 150us in kernel mode
CACHE: Requested read for buffer (0:100)
CACHE: Buffer (0:100) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:100)
DRIVER: Using LOOK strategy
 Direction: OUT
 Current buffer: (0:100)
 Buffer queue: [ ]
DRIVER: Moving from track 0 to 0 in 0us
 Direct move time: 0us
 Move time with rewind: 0us
DRIVER: Sector access in 4016us
SCHEDULER: Next driver interruption at 4166us
SCHEDULER: Block user process "yyy"

SCHEDULER: 150us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in user mode
```

```
SCHEDULER: User process "qqq" invoked read() for buffer
(0:110)

SCHEDULER: 150us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in kernel mode
... User process "qqq" spent 150us in kernel mode
CACHE: Requested read for buffer (0:110)
CACHE: Buffer (0:110) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (0:110)
SCHEDULER: Block user process "qqq"

SCHEDULER: 300us (NEXT ITERATION)
SCHEDULER: Running user process "eee" in user mode
SCHEDULER: User process "eee" invoked read() for buffer
(3:1500)

SCHEDULER: 300us (NEXT ITERATION)
SCHEDULER: Running user process "eee" in kernel mode
... User process "eee" spent 150us in kernel mode
CACHE: Requested read for buffer (3:1500)
CACHE: Buffer (3:1500) not found
CACHE: Getting free buffer
CACHE: Using LRU strategy
 Left segment: [ ]
 Right segment: [ ]
```

```
CACHE: Requesting driver read
DRIVER: Requested IO (read) for buffer (3:1500)
SCHEDULER: Block user process "eee"

SCHEDULER: 450us (NEXT ITERATION)
SCHEDULER: Nothing to do for 3716us

SCHEDULER: 4166us (NEXT ITERATION)

<<< Begin driver interruption at 4166us
DRIVER: Completed IO (read) for buffer (0:100)
CACHE: Buffer (0:100) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "yyy"
DRIVER: Using LOOK strategy
 Direction: OUT
 Current buffer: (3:1500)
 Buffer queue: [ (0:110) ]
DRIVER: Moving from track 0 to 3 in 1500us
 Direct move time: 1500us
 Move time with rewind: 1510us
DRIVER: Sector access in 5516us
SCHEDULER: Next driver interruption at 9732us
... Driver interruption spent 50us
>>> End driver interruption

SCHEDULER: 4216us (NEXT ITERATION)
SCHEDULER: Running user process "yyy" in user mode
```

```
... User process "yyy" spent 5516us in user mode

<<< Begin driver interruption at 9732us
DRIVER: Completed IO (read) for buffer (3:1500)
CACHE: Buffer (3:1500) added to cache
CACHE: Using LRU strategy
 Left segment: [ (3:1500) (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "eee"
DRIVER: Using LOOK strategy
 Direction: IN
 Current buffer: (0:110)
 Buffer queue: [ ]
DRIVER: Moving from track 3 to 0 in 10us
 Direct move time: 1500us
 Move time with rewind: 10us
DRIVER: Sector access in 4026us
SCHEDULER: Next driver interruption at 15292us
... Driver interruption spent 50us
>>> End driver interruption

... User process "yyy" spent 1484us in user mode
SCHEDULER: User process "yyy" exited

SCHEDULER: 11266us (NEXT ITERATION)
SCHEDULER: Running user process "eee" in user mode
... User process "eee" spent 4026us in user mode

<<< Begin driver interruption at 15292us
DRIVER: Completed IO (read) for buffer (0:110)
```

```
CACHE: Buffer (0:110) added to cache
CACHE: Using LRU strategy
 Left segment: [ (0:110) (3:1500) (0:100) ]
 Right segment: [ ]
SCHEDULER: Wake up user process "qqq"
DRIVER: Nothing to do
... Driver interruption spent 50us
>>> End driver interruption

... User process "eee" spent 2974us in user mode
SCHEDULER: User process "eee" exited

SCHEDULER: 18316us (NEXT ITERATION)
SCHEDULER: Running user process "qqq" in user mode
... User process "qqq" spent 7000us in user mode
SCHEDULER: User process "qqq" exited

SCHEDULER: 25316us (NEXT ITERATION)
SCHEDULER: All user processes finished, flushing buffer
cache
CACHE: Flushing buffers
CACHE: Successfully flushed

SCHEDULER: 25316us (NEXT ITERATION)
SCHEDULER: All user processes finished, flushing buffer
cache
CACHE: Flushing buffers
CACHE: Already flushed
SCHEDULER: Buffer cache flushed, exiting
```