

# 学生实验报告

开课学院及实验室：计算机科学与工程实验室电子楼 412

2023 年 05 月 09 日

学院	计算机科学与网络工程学院	年级/专业/班		姓名		学号	
实验课程名称	计算机系统结构与操作系统					成绩	
实验项目名称	实验二 操作系统算法实验					指导老师	

## 一、实验目的

掌握操作系统常用算法和功能，并编程实现。包括：银行家算法、页面置换算法、文件管理、磁盘调度算法等。

## 二、实验设备和资料

- 1、微型计算机：Linux 操作系统。
- 2、《操作系统实验指导书》的实验二。

## 三、实验内容与运行结果

### 1. 银行家算法

- Linux 环境下编译运行程序
- 程序数据结构变量名与教材一致
- 可支持不同个数进程与不同个数资源
- 验证教材例子

#### ★ 源代码

```
/*
 * @Author: 
 * @Date: 2023-04-24 11:36:37
 * @LastEditTime: 2023-05-16 17:33:47
 * @FilePath: /源码/Banker.cpp
 * @Description: 银行家算法
 * Copyright (c) 2023 by GZHU-FWM, All Rights Reserved.
 */

#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
#include <iomanip>
#include <vector>

using namespace std;
```

```

class Banker
{
public:
    ~Banker(); // 析构
    void InitBanker(const string &fname); // 初始化各项数据
    int SetRequest(int i); // 设置请求向量
    bool SafeCheck(); // 银行家算法检查安全性
    bool RequestSafe(int i); // 检查进程 Pi 请求是否安全
    void PrintForm(); // 打印当前状态表格
    void PrintList(); // 打印安全序列

private:
    int m; // 资源种类
    int n; // 进程数
    int **Max; // 最大需求矩阵
    int **Need; // 需求矩阵
    int *Request; // 请求向量
    int *Available; // 可利用资源向量
    int **Allocation; // 已分配矩阵
    vector<int> SafeList; // 安全序列

    int RequestCheck(int i); // 检查进程资源请求是否合法(合法≠安全)
};

Banker::~Banker()
{
    delete[] Available;
    delete[] Request;
    for (int i = 0; i < n; i++)
    {
        delete[] Max[i];
        delete[] Allocation[i];
        delete[] Need[i];
    }
    delete[] Max;
    delete[] Allocation;
    delete[] Need;
    m = -1;
    n = -1;
    SafeList.clear();
}

// 初始化各项数据
void Banker::InitBanker(const string &fname)
{

```

```

string line;
ifstream ifs;
stringstream ss;

cout << "\nCopyright (c) 2023 by FWM-██████████, All Rights
Reserved.\n"
    << endl;

// 打开文件
ifs.open(fname);
if (!ifs.is_open())
{
    cout << "文件打开失败!" << endl;
    return;
}
cout << "读取文件内容如下: \n";

// 读取文件
// 资源种类数
getline(ifs, line);
m = stoi(line);
Available = new int[m];
vector<int> temp(m, 0);

// 读取进程数
getline(ifs, line);
n = stoi(line);
// 初始化动态数组
Request = new int[m];
Max = new int * [n];
Allocation = new int * [n];
Need = new int * [n];
for (int i = 0; i < n; i++)
{
    Max[i] = new int[m];
    Allocation[i] = new int[m];
    Need[i] = new int[m];
}

// 读取各进程最大需求
cout << "=====" << endl;
cout << "最大需求 Max: \n";
for (int i = 0; i < n; i++)
{

```

```

getline(ifs, line);
ss.str(line);
cout << (char)('a' + i) << ": ";
for (int j = 0; j < m; j++)
{
    ss >> Max[i][j];
    cout << Max[i][j] << " ";
}
ss.clear();
cout << endl;
}

// 读取各进程已分配资源数
cout << "=====" << endl;
cout << "已分配资源数 Allocation: \n";
for (int i = 0; i < n; i++)
{
    getline(ifs, line);
    ss.str(line);
    cout << (char)('a' + i) << ": ";
    for (int j = 0; j < m; j++)
    {
        ss >> Allocation[i][j];
        Need[i][j] = Max[i][j] - Allocation[i][j];
        temp[j] += Allocation[i][j];
        cout << Allocation[i][j] << " ";
    }
    ss.clear();
    cout << endl;
}

// 读取各类资源总量
cout << "=====" << endl;
cout << "各类资源总量: \n";
getline(ifs, line);
ss.str(line);
for (int i = 0; i < m; i++)
{
    cout << 'r' + to_string(i + 1) << ": ";
    ss >> Available[i];
    cout << Available[i] << " ";
    Available[i] -= temp[i];
}

```

```

cout << endl;
ss.clear();

// 关闭文件
ifs.close();
}
// 银行家算法安全检测
bool Banker::SafeCheck()
{
    bool flag = true;           // 标志, 是否满足 step2 条件
    int *Work = new int[m];      // 工作向量 (就是个 temp)
    bool *Finish = new bool[n]; // 标志进程是否已经运行
    for (int i = 0; i < n; i++)
        Finish[i] = false;
    SafeList.clear();

    // step0: Work = Available
    for (int i = 0; i < m; i++)
        Work[i] = Available[i];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            flag = true;
            // step1: 检查进程 Pj 是否已经运行
            if (Finish[j])
                continue;
            // step2: 检查进程 Pj 是否 Need <= Available
            for (int k = 0; k < m; k++)
            {
                if (Need[j][k] <= Work[k])
                {
                    continue;
                }
                else
                {
                    flag = false;
                    break;
                }
            }
            if (!flag)
                continue;
            // step3: 分配资源, 运行 Pj, 回收资源
            for (int k = 0; k < m; k++)

```

```

        Work[k] = Work[k] + Allocation[j][k];
        Finish[j] = true;
        // 将 Pj 添加进安全序列
        SafeList.push_back(j);
        // 回到 step1
        break;
    }
}
// step4: 检查所有进程的 Finish[i]=true, 是则安全, 反之寄
for (int i = 0; i < n; i++)
{
    if (Finish[i])
        continue;
    else
    {
        delete[] Work;
        delete[] Finish;
        SafeList.clear();
        return false;
    }
}
delete[] Work;
delete[] Finish;
return true;
}
// 进程 Pi 发出请求
int Banker::SetRequest(int i)
{
    cout << "请输入请求资源量: ";
    for (int i = 0; i < m; i++)
        cin >> Request[i];
    cout << endl;
    // 检查 Request i 是否合法
    return RequestCheck(i);
}
// 检查对进程 Pi 的请求是否合法
int Banker::RequestCheck(int i)
{
    for (int j = 0; j < m; j++)
    {
        // step1: 检查 Request<=Need
        if (!(Request[j] <= Need[i][j]))
            return -1;
        // step2: 检查 Request<=Available

```



```

        else if (!(Request[j] <= Available[j]))
            return -2;
    }
    return 0;
}
// 检查对进程 Pi 的请求是否安全
bool Banker::RequestSafe(int i)
{
    // 试分配
    for (int j = 0; j < m; j++)
    {
        Allocation[i][j] += Request[j];
        Need[i][j] -= Request[j];
        Available[j] -= Request[j];
    }
    PrintForm();
    // 安全性检测
    if (SafeCheck())
        return true;
    else
    {
        // 不安全, 还原
        for (int j = 0; j < m; j++)
        {
            Allocation[i][j] -= Request[j];
            Need[i][j] += Request[j];
            Available[j] += Request[j];
        }
        return false;
    }
}
// 打印状态表
void Banker::PrintForm()
{
    cout << "Available = { ";
    for (int i = 0; i < m; i++)
    {
        cout << Available[i];
        if (i != m - 1)
            cout << ", ";
    }
    cout << " }\n"
         << endl;
    cout << "Pi"

```

```

    << " | "
    << "Max \t|"
    << " Allo\t\t|"
    << " Need\t\t|" << endl;
for (int i = 0; i < n; i++)
{
    cout << (char)('a' + i) << " | ";
    for (int j = 0; j < m; j++)
        cout << Max[i][j] << " ";
    cout << "\t| ";
    for (int j = 0; j < m; j++)
        cout << Allocation[i][j] << " ";
    cout << "\t| ";
    for (int j = 0; j < m; j++)
        cout << Need[i][j] << " ";
    cout << "\t| ";
    cout << endl;
}
}
// 打印安全序列
void Banker::PrintList()
{
    cout << "安全序列为: ";
    cout << "< ";
    for (int i = 0; i < n; i++)
    {
        if (!SafeList.empty())
            cout << (char)('a' + SafeList[i]);
        else
            return;
        if (i != n - 1)
            cout << ", ";
    }
    cout << " >" << endl;
}

int main(int argc, char *argv[])
{
    // 读取参数
    if (argc != 2)
    {
        cout << "ERROR: 参数输入错误! 请检查后重试。" << endl;
        exit(-1);
    }
}

```



```

string fname = argv[1];
cout << "输入参数为: " << fname << endl;

char val = 'y';

// 初始化
Banker B;
B.InitBanker(fname);
cout << "\n===== " << endl;
cout << "T0 时刻: \n";
B.PrintForm();
if (B.SafeCheck())
{
    cout << "安全\n";
    B.PrintList();
}
else
{
    cout << "不安全!\n";
    exit(-1);
}
while (val == 'y')
{
    char c = '?';
    cout << "\n===== " << endl;
    cout << "请输入申请资源的进程号: ";
    cin >> c;
    switch (B.SetRequest(c - 'a'))
    {
        case -1:
            cout << "进程请求不合法, 请求大于需求! " << endl;
            break;
        case -2:
            cout << "进程请求不合法, 请求大于系统拥有! " << endl;
            break;
        case 0:
            cout << "\n===== " << endl;
            cout << "进程请求合法" << endl;
            if (B.RequestSafe(c - 'a'))
            {
                cout << "请求安全! " << endl;
                B.PrintList();
            }
            else

```

```

    {
        cout << "请求不安全！" << endl;
        cout << "已恢复安全状态！\n\n";
    }
    break;
}
cout << "\n===== " << endl;
cout << "是否继续(y/n): ";
cin >> val;
}
system("pause");
return 0;
}

```

#### ★ 测试数据文件（教材例子）

```

3
4
3 2 2
6 1 3
3 1 4
4 2 2
1 0 0
5 1 1
2 1 1
0 0 2
9 3 6

```

#### ★ 测试数据文件解析

3	←	资源数量
4	←	进程数量
3 2 2	←	进程需要资源数 Max
6 1 3	←	
3 1 4	←	
4 2 2	←	
1 0 0	←	进程拥有资源数 Allocation
5 1 1	←	
2 1 1	←	
0 0 2	←	
9 3 6	←	各类资源总数

## 运行截图

```
fwm-0100@ubuntu-18: ~/OS/实验2/源码
fwm-0100@ubuntu-18:~/OS/实验2/源码$ g++ Banker.cpp -o Banker
fwm-0100@ubuntu-18:~/OS/实验2/源码$ ./Banker banker_data
输入参数为：banker_data

Copyright (c) 2023 by FWM-32116160100 , All Rights Reserved.

读取文件内容如下：
=====
最大需求Max：
a: 3 2 2
b: 6 1 3
c: 3 1 4
d: 4 2 2
=====
已分配资源数Allocation：
a: 1 0 0
b: 5 1 1
c: 2 1 1
d: 0 0 2
=====
各类资源总量：
r1: 9 r2: 3 r3: 6
=====
T0时刻：
Available = { 1, 1, 2 }

Pi | Max | Allo | Need |
a | 3 2 2 | 1 0 0 | 2 2 2 |
b | 6 1 3 | 5 1 1 | 1 0 2 |
c | 3 1 4 | 2 1 1 | 1 0 3 |
d | 4 2 2 | 0 0 2 | 4 2 0 |
安全
安全序列为：< b, a, c, d >

=====
请输入申请资源的进程号：a
请输入请求资源量：1 0 1

=====
进程请求合法
Available = { 0, 1, 1 }

Pi | Max | Allo | Need |
a | 3 2 2 | 2 0 1 | 1 2 1 |
b | 6 1 3 | 5 1 1 | 1 0 2 |
c | 3 1 4 | 2 1 1 | 1 0 3 |
d | 4 2 2 | 0 0 2 | 4 2 0 |
请求不安全！
已恢复安全状态！

=====
是否继续(y/n)：y
```

## 2. 常用页面置换算法模拟

- 输入数据从文本读入
- 实现最佳淘汰算法（OPT），先进先出的算法（FIFO），最近最久未使用算法（LRU）
- 输出访问顺序和命中率结果

## 源代码

```
/*
 * @Author: 
 * @Date: 2023-04-24 11:49:16
 * @LastEditTime: 2023-05-03 10:28:45
 * @FilePath: /源码/PageSwapSim.cpp
 * @Description: 页面置换算法模拟
 * Copyright (c) 2023 by GZHU-FWM, All Rights Reserved.
```

```

*/
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
#include <iomanip>
#include <limits.h>

using namespace std;

// 输出文字颜色相关的宏
#define NONE "\033[m"
#define YELLOW "\033[1;33m"
#define LIGHT_BLUE "\033[1;34m"
#define LIGHT_CYAN "\033[1;36m"

#define MaxSize 10 // 队列最大长度
#define ElemType int // 队列元素的数据类型

// ----- 循环队列 -----
typedef struct
{
    ElemType *base; // 动态数组指针
    int front;      // 头指针，指向队头
    int rear;       // 尾指针，指向队尾
} SqQueue;

bool InitQueue(SqQueue &Q);           // 初始化队列
bool DestroyQueue(SqQueue &Q);        // 销毁队列
int QueueLength(SqQueue Q);           // 队列求长
bool EnQueue(SqQueue &Q, ElemType e); // 入队
bool DeQueue(SqQueue &Q, ElemType &e); // 出队

// ----- 测试数据 -----
class PageData
{
public:
    PageData();
    ~PageData();
    void InitData(const string &fname); // 从文件中读取数据初始化数据成员
    void PrintData();                  // 打印初始化后的数据

    int blockNum; // 内存块数量
    int pageNum;  // 页面数量

```

```

    int *pageList; // 页面号序列
} pd;

// ----- 记录过程 -----
class PageTrack
{
private:
    int **optTrack; // 记录 opt 过程
    int **fifoTrack; // 记录 fifo 过程
    int **lruTrack; // 记录 lru 过程
    bool **faults; // 记录缺页情况
    int pos[3]; // 数组指针, [0]opt, [1]fifo...
    double result[3]; // 命中率结果

public:
    PageTrack();
    ~PageTrack();
    void InitTrack(); // 初始化结构体的数据成员
    void UpdateTrack(int type, int num, int page, int flag); // 每读取到一个新的页面记录一次
    void PrintTrack(int type); // 打印过程
    void HitRate(int type, int miss); // 计算命中率
} track;

// ----- 公共操作 -----
bool CheckFault(int *block, int key); // 用于检查是否缺页

// ----- OPT 算法部分 -----
void OPT_FRead(int *block, int &pagePos); // 内存为空时读入页
int NextVis(int pagePos, int num); // 计算页面的下一次访问的时间
int OPT_Replace(int pagePos, int *block); // 选择内存块进行替换
void OPT(); // OPT 算法

// ----- FIFO 算法部分 -----
void FIFO_FRead(int *block, int &pagePos, SqQueue &sq); // 内存为空时读入页
int FIFO_Replace(int *block, SqQueue &sq); // 选择内存块进行替换
void FIFO(); // FIFO 算法

// ----- LRU 算法部分 -----
void LRU_FRead(int *block, int &pagePos, SqQueue &sq); // 内存为空时读入

```

```

int LRU_Replace(int *block, SqQueue &sq); // 选择内存块进行替换
void LRU_Flash(int pagePos, SqQueue &sq, SqQueue &temSq); // 更新访问时间
void LRU(); // LRU 算法

int main(int argc, char *argv[])
{
    // 读取参数
    if (argc != 2)
    {
        cout << "ERROR: 参数输入错误！请检查后重试。" << endl;
        exit(-1);
    }
    string fname = argv[1];
    cout << "输入参数为: " << fname << endl;

    // 初始化
    pd.InitData(fname);
    pd.PrintData();
    track.InitTrack();

    // 运行算法
    OPT();
    FIFO();
    LRU();

    // 分别打印每个算法的运行过程以及结果
    for (int i = 0; i < 3; i++)
        track.PrintTrack(i);

    exit(0);
}

// ----- 循环队列 -----
// 初始化队列
bool InitQueue(SqQueue &Q)
{
    Q.base = new ElemType[MaxSize]; // 分配动态空间
    // 如果 base 指针为 NULL, 代表空间分配失败
    if (!Q.base)
    {
        cout << "队列初始化失败!" << endl;
        return false;
    }
    Q.front = Q.rear = 0; // 空间分配成功, 初始化队头与队尾指针
}

```



```

        return true;
    }
    // 销毁队列
    bool DestroyQueue(SqQueue &Q)
    {
        delete Q.base;          // 释放动态分配的空间
        Q.base = NULL;          // 数组指针置空
        Q.front = Q.rear = 0;    // 头尾置 0, 队列空
        return true;
    }
    // 队列求长
    int QueueLength(SqQueue Q)
    {
        return (Q.rear - Q.front + MaxSize) % MaxSize;
    }
    // 入队
    bool EnQueue(SqQueue &Q, ElemType e)
    {
        // 判断队是否已满, 满则返回错误
        if ((Q.rear + 1) % MaxSize == Q.front)
        {
            cout << "队满!!!" << endl;
            return false;
        }
        Q.base[Q.rear] = e;      // 元素 e 入队尾
        Q.rear = (Q.rear + 1) % MaxSize; // 根据循环队列逻辑, 使尾指针逻辑后移
        return true;
    }
    // 出队
    bool DeQueue(SqQueue &Q, ElemType &e)
    {
        // 判断队是否已空, 空则返回错误
        if (Q.front == Q.rear)
        {
            cout << "队空!!!" << endl;
            return false;
        }
        e = Q.base[Q.front];
        Q.front = (Q.front + 1) % MaxSize; // 根据循环队列逻辑, 使头指针逻辑后移
        return true;
    }
    // ----- 测试数据 -----

```

---

```

PageData::PageData()
{
    int blockNum = -1;
    int pageNum = -1;
    int *pageList = NULL;
}
PageData::~PageData()
{
    blockNum = -1;
    pageNum = -1;
    delete[] pageList;
}
// 从文件中读取数据初始化
void PageData::InitData(const string &fname)
{
    string line;
    ifstream ifs;
    stringstream ss;

    cout << "\nCopyright (c) 2023 by FWM-██████████, All Rights Reserved.\n"
         << endl;

    // 打开文件
    ifs.open(fname);
    if (!ifs.is_open())
    {
        cout << "文件打开失败!" << endl;
        return;
    }

    // 读取文件
    // i 为行号
    for (int i = 1; getline(ifs, line); i++)
    {
        switch (i)
        {
            case 1:
                // 读取块号
                blockNum = stoi(line);
                break;

            case 2:
                // 读取页面序列数量

```

```

        pageNum = stoi(line);
        if (pageNum <= 0)
        {
            cout << "文件数据有误！" << endl;
            return;
        }
        // 创建页面序列的存储空间
        pageList = new int[pageNum];
        break;

    case 3:
        int k = 0;    // pageList 指针
        int num = -1; // 存储每次从 ss 里读取的数
        // 初始化 ss
        ss.str(line);
        while (ss >> num)
        {
            // 存储到 pageList
            pageList[k] = num;
            // 指针后移
            k++;
        }
        break;
    }
    // 关闭文件
    ifs.close();
}
// 打印初始化后的数据
void PageData::PrintData()
{
    cout << "文件内容: \n";
    // 块数量
    cout << blockNum << endl;
    // 读取页面序列数量
    cout << pageNum << endl;
    for (int i = 0; i < pd.pageNum; i++)
        cout << pageList[i] << " ";
    cout << endl;
}

// ----- 记录过程 -----
PageTrack::PageTrack()
{

```

```

int **optTrack = NULL;
int **fifoTrack = NULL;
int **lruTrack = NULL;
int **faults = NULL;
}
PageTrack::~PageTrack()
{
    for (int i = 0; i < pd.blockNum; i++)
    {
        if (i < 3)
            delete[] faults[i];
        delete[] optTrack[i];
        delete[] fifoTrack[i];
        delete[] lruTrack[i];
    }
    delete[] optTrack;
    delete[] fifoTrack;
    delete[] lruTrack;
    delete[] faults;
}
// 初始化
void PageTrack::InitTrack()
{
    // 创建三个[blockNum][pageNum]的数组
    optTrack = new int *[pd.blockNum];
    fifoTrack = new int *[pd.blockNum];
    lruTrack = new int *[pd.blockNum];
    faults = new bool *[3];

    for (int i = 0; i < pd.blockNum; i++)
    {
        if (i < 3)
        {
            faults[i] = new bool[pd.pageNum];
            pos[i] = 0;
            result[i] = -1;
        }
        optTrack[i] = new int[pd.pageNum];
        fifoTrack[i] = new int[pd.pageNum];
        lruTrack[i] = new int[pd.pageNum];
        for (int k = 0; k < pd.pageNum; k++)
        {
            if (i < 3)
                faults[i][k] = 0;
        }
    }
}

```

```

        optTrack[i][k] = -1;
        fifoTrack[i][k] = -1;
        lruTrack[i][k] = -1;
    }
}
// 每读取到一个新的页面记录一次
void PageTrack::UpdateTrack(int type, int num, int page, int flag)
{
    // flag:1 更改 0 复制（没有发生页面置换）
    switch (type)
    {
        case 0:
            // 拷贝*Track[type][pos-1]
            if (pos[type] - 1 >= 0)
                for (int i = 0; i < pd.blockNum; i++)
                    optTrack[i][pos[type]] = optTrack[i][pos[type] - 1];
            // 更改
            if (flag)
                optTrack[num][pos[type]] = page;
            break;
        case 1:
            // 拷贝*Track[type][pos-1]
            for (int i = 0; i < pd.blockNum; i++)
                if (pos[type] - 1 >= 0)
                    fifoTrack[i][pos[type]] = fifoTrack[i][pos[type] - 1];
            if (flag)
                fifoTrack[num][pos[type]] = page;
            break;
        case 2:
            // 拷贝*Track[type][pos-1]
            for (int i = 0; i < pd.blockNum; i++)
                if (pos[type] - 1 >= 0)
                    lruTrack[i][pos[type]] = lruTrack[i][pos[type] - 1];
            if (flag)
                lruTrack[num][pos[type]] = page;
            break;
    }
    if (flag)
        faults[type][pos[type]] = 1;
    pos[type]++;
}
// 打印过程
void PageTrack::PrintTrack(int type)

```

```

{
    switch (type)
    {
        case 0:
            cout << LIGHT_CYAN << "\nOPT\n";
            cout << NONE << "序列  |";
            for (int k = 0; k < pd.pageNum; k++)
                cout << setw(2) << pd.pageList[k] << " ";
            cout << endl;
            for (int i = 0; i < pd.blockNum; i++)
            {
                cout << "内存块" << i << "|";
                for (int j = 0; j < pd.pageNum; j++)
                {
                    if (optTrack[i][j] == -1)
                    {
                        cout << " ";
                        continue;
                    }
                    cout << setw(2) << optTrack[i][j] << " ";
                }
                cout << endl;
            }
            break;
        case 1:
            cout << LIGHT_CYAN << "\nFIFO\n";
            cout << NONE << "序列  |";
            for (int k = 0; k < pd.pageNum; k++)
                cout << setw(2) << pd.pageList[k] << " ";
            cout << endl;
            for (int i = 0; i < pd.blockNum; i++)
            {
                cout << "内存块" << i << "|";
                for (int j = 0; j < pd.pageNum; j++)
                {
                    if (fifoTrack[i][j] == -1)
                    {
                        cout << " ";
                        continue;
                    }
                    cout << setw(2) << fifoTrack[i][j] << " ";
                }
                cout << endl;
            }
    }
}

```



```

        break;
    case 2:
        cout << LIGHT_CYAN << "\nLRU\n";
        cout << NONE << "序列  |";
        for (int k = 0; k < pd.pageNum; k++)
            cout << setw(2) << pd.pageList[k] << " ";
        cout << endl;
        for (int i = 0; i < pd.blockNum; i++)
        {
            cout << "内存块" << i << "|";
            for (int j = 0; j < pd.pageNum; j++)
            {
                if (lruTrack[i][j] == -1)
                {
                    cout << " ";
                    continue;
                }
                cout << setw(2) << lruTrack[i][j] << " ";
            }
            cout << endl;
        }
        break;
    }
    cout << "缺页  |";
    string sig = "v";
    for (int i = 0; i < pd.pageNum; i++)
    {
        if (faults[type][i])
            cout << LIGHT_BLUE << setw(4) << sig << " ";
        else
            cout << " ";
    }
    cout << "\n"
        << YELLOW << "命中率 : " << setw(2) << result[type] << "%" << endl;
}
// 计算命中率
void PageTrack::HitRate(int type, int miss)
{
    double lackRate = (double)miss / (double)pd.pageNum;
    result[type] = (1.0 - lackRate) * 100.0;
}

// ----- 公共操作 -----
// 用于检查是否缺页(key 是否位于 block[]内)

```

```

bool CheckFault(int *block, int key)
{
    for (int i = 0; i < pd.blockNum; i++)
    {
        // 需要使用的页面在内存中存在
        // 不缺页返回 false
        if (block[i] == key)
            return false;
    }
    // 缺页返回 true
    return true;
}

// ----- OPT 算法部分 -----
// 内存为空时读入页
void OPT_FRead(int *block, int &pagePos)
{
    // 初始化
    int i = 0;
    pagePos = 0;
    for (int i = 0; i < pd.blockNum; i++)
        block[i] = -1;

    while (i < pd.blockNum)
    {
        // 避免重复读入
        if (CheckFault(block, pd.pageList[pagePos]))
        {
            block[i] = pd.pageList[pagePos];
            // 有更改
            track.UpdateTrack(0, i, block[i], 1);
            i++;
            pagePos++;
        }
        else
        {
            // 没有更改，直接复制
            track.UpdateTrack(0, 0, 0, 0);
            pagePos++;
        }
    }
}

// 计算页面的下一次访问的时间（用页表中相距的距离表示）
int NextVis(int pagePos, int num)

```

```

{
    // pagePos page 序列当前位置
    // num 需要计算的页号
    int count = 0;
    for (int i = pagePos; i < pd.pageNum; i++)
    {
        if (pd.pageList[i] == num)
            return count;
        count++;
    }
    return INT_MAX;
}

// 用于选择内存中哪一个内存块进行替换
int OPT_Replace(int pagePos, int *block)
{
    // max temp 为该页未来下一次访问的时间
    // num 下一次访问时间最久的页面号对应的内存块号
    int max, temp, num;
    max = temp = num = INT_MIN;
    // 选择未来最久不会被访问的页淘汰
    for (int i = 0; i < pd.blockNum; i++)
    {
        temp = NextVis(pagePos, block[i]);
        if (temp > max)
        {
            max = temp;
            num = i;
        }
    }
    return num;
}

// OPT 算法
void OPT()
{
    int *block = new int[pd.blockNum]; // 内存块
    int miss = 0;                       // 缺页次数
    int pagePos = 0;                    // page 序列当前位置
    double lackRate;                   // 缺页中断率

    // 一开始内存为空, 读入页
    OPT_FRead(block, pagePos);
    // 更新缺页次数
    miss = pd.blockNum;

```

```

// 不断读入
while (pagePos < pd.pageNum)
{
    if (CheckFault(block, pd.pageList[pagePos]))
    {
        // 缺页
        miss++;
        // 找到要淘汰的内存块号
        int num = OPT_Replace(pagePos, block);
        if (num == -1 || num > pd.blockNum)
        {
            cout << "OPT_Replace()出错! 程序中止" << endl;
            exit(-1);
        }
        block[num] = pd.pageList[pagePos];
        // 有更改
        track.UpdateTrack(0, num, block[num], 1);
    }
    else
    {
        // 没有更改, 直接复制
        track.UpdateTrack(0, 0, 0, 0);
    }
    pagePos++;
}
// 计算缺页率, 并存储到数组 result 中
track.HitRate(0, miss);
delete[] block;
}

// ----- FIFO 算法部分 -----
// 内存为空时读入页
void FIFO_FRead(int *block, int &pagePos, SqQueue &sq)
{
    // 初始化
    int i = 0;
    pagePos = 0;
    for (int i = 0; i < pd.blockNum; i++)
        block[i] = -1;
    while (i < pd.blockNum)
    {
        // 避免重复读入
        if (CheckFault(block, pd.pageList[pagePos]))
        {

```

```

        // 读入
        block[i] = pd.pageList[pagePos];
        // 记录, 有更改
        track.UpdateTrack(1, i, block[i], 1);
        // !! 入队列
        EnQueue(sq, block[i]);
        i++;
        pagePos++;
    }
    else
    {
        // 记录, 无更改
        track.UpdateTrack(1, 0, 0, 0);
        pagePos++;
    }
}

// 用于选择内存中哪一个内存块进行替换
int FIFO_Replace(int *block, SqQueue &sq)
{
    int e; // 被淘汰的页号
    if (!DeQueue(sq, e))
    {
        printf("出队失败! 队空!!!\n");
        return -1;
    }
    for (int i = 0; i < pd.blockNum; i++)
    {
        if (block[i] == e)
            return i;
    }
    return -1;
}

// FIFO 算法
void FIFO()
{
    SqQueue sq;
    InitQueue(sq);
    int *block = new int[pd.blockNum]; // 内存块
    int miss = 0;                       // 缺页次数
    int pagePos = 0;                    // page 序列当前位置
    double lackRate;                    // 缺页中断率

    // 一开始内存为空, 读入页

```

```

FIFO_FRead(block, pagePos, sq);
// 更新缺页次数
miss = pd.blockNum;

while (pagePos < pd.pageNum)
{
    if (CheckFault(block, pd.pageList[pagePos]))
    {
        // 缺页
        miss++;
        // 找到要淘汰的页框号
        int num = FIFO_Replace(block, sq);
        block[num] = pd.pageList[pagePos];
        // 入队
        EnQueue(sq, pd.pageList[pagePos]);
        // 记录, 有更改
        track.UpdateTrack(1, num, block[num], 1);
    }
    else
    {
        // 记录, 无更改
        track.UpdateTrack(1, 0, 0, 0);
    }
    pagePos++;
}
track.HitRate(1, miss);
delete[] block;
DestroyQueue(sq);
}

// ----- LRU 算法部分 -----
// 内存为空时读入页
void LRU_FRead(int *block, int &pagePos, SqQueue &sq)
{
    // 初始化
    int i = 0;
    pagePos = 0;
    for (int i = 0; i < pd.blockNum; i++)
        block[i] = -1;

    while (i < pd.blockNum)
    {
        // 避免重复读入
        if (CheckFault(block, pd.pageList[pagePos]))

```



```

    {
        block[i] = pd.pageList[pagePos];
        EnQueue(sq, block[i]);
        track.UpdateTrack(2, i, block[i], 1);
        i++;
        pagePos++;
    }
    else
    {
        track.UpdateTrack(2, 0, 0, 0);
        pagePos++;
    }
}
}
// 用于选择内存中哪一个内存块进行替换
int LRU_Replace(int *block, SqQueue &sq)
{
    int e; // 被淘汰的页号
    if (!DeQueue(sq, e))
        return -1;
    for (int i = 0; i < pd.blockNum; i++)
    {
        if (block[i] == e)
            return i;
    }
    return -1;
}
// 更新访问时间
void LRU_Flash(int pagePos, SqQueue &sq, SqQueue &temSq)
{
    int len; // 队列长度
    int pageNum; // 页号
    len = QueueLength(sq);
    for (int i = 0; i < len; i++)
    {
        // 出队
        if (!DeQueue(sq, pageNum))
            exit(-1);
        // 避免复制到需要更新访问时间的页号
        if (pageNum == pd.pageList[pagePos])
            continue;
        // 复制
        EnQueue(temSq, pageNum);
    }
}

```

```

}
// 更新访问时间
EnQueue(temSq, pd.pageList[pagePos]);
// 将更新完成的队列与原来的队列交换
SqQueue temp = temSq;
temSq = sq;
sq = temp;
// 记录, 无更改
track.UpdateTrack(2, 0, 0, 0);
}
// LRU 算法
void LRU()
{
    int miss = 0; // 缺页次数
    int *block = new int[pd.blockNum]; // 内存块
    int pagePos = 0; // page 序列当前位置
    double lackRate; // 缺页中断率
    SqQueue sq; // 队列, 记录访问时间
    SqQueue temSq; // 用于更新访问时间的临时队列

    InitQueue(sq);
    InitQueue(temSq);

    // 一开始内存为空, 读入页
    LRU_FRead(block, pagePos, sq);
    // 更新缺页次数
    miss = pd.blockNum;

    while (pagePos < pd.pageNum)
    {
        if (CheckFault(block, pd.pageList[pagePos]))
        {
            // 缺页
            miss++;
            // 找到要淘汰的页
            int num = LRU_Replace(block, sq);
            block[num] = pd.pageList[pagePos];
            EnQueue(sq, pd.pageList[pagePos]);
            track.UpdateTrack(2, num, block[num], 1);
        }
        else
        {
            // 不缺页
            // 需要更改访问时间

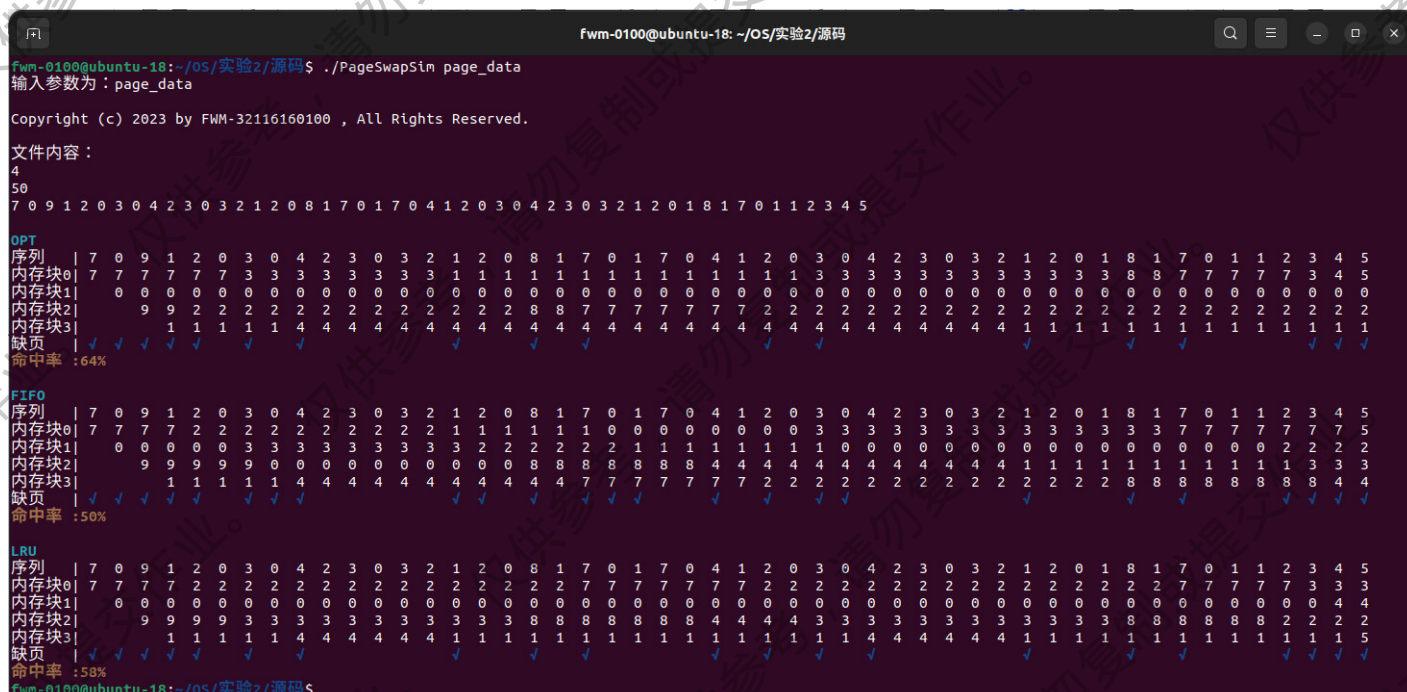
```

```

        LRU_Flash(pagePos, sq, temSq);
    }
    pagePos++;
}
// 计算缺页中断率
track.HitRate(2, miss);
delete[] block;

// 销毁队列
DestroyQueue(sq);
DestroyQueue(temSq);
}

```



- 1) 分别利用 fwrite 接口和 write 接口追写文件 100 万次, 内容 “Good Moring,FWM”, 使用高精度时间函数计算写文件 100 万次所需要的时间。

```
/*
 * @Author: [REDACTED]
 * @Date: 2023-05-07 17:07:03
 * @LastEditTime: 2023-05-08 15:06:17
 * @FilePath: /源码/FileMgmt.cpp
 * @Description: 文件管理实验 - fwrite/write
 * Copyright (c) 2023 by GZHU-FWM, All Rights Reserved.
 */
```

```

#include <iostream>
#include <cstring>
#include <cstdio>
#include <chrono>
#include <fcntl.h>
#include <unistd.h>

using namespace std;
using namespace std::chrono;

#define COUNT 1000000

void fwriteTime(const char *str, const char *fname, int count)
{
    // 以追加方式打开文件
    FILE *fp = fopen(fname, "a");
    if (!fp)
    {
        cout << "文件打开失败!" << endl;
        exit(-1);
    }
    // 开始计时
    auto startTime = high_resolution_clock::now();
    // 写入文件
    for (int i = 0; i < count; i++)
        fwrite(str, sizeof(char), strlen(str), fp);
    // 结束计时
    auto endTime = high_resolution_clock::now();
    cout << "fwriteTime:\t" << duration_cast<milliseconds>(endTime -
    startTime).count() << " ms" << endl;
    // 关闭文件
    fclose(fp);
}

void writeTime(const char *str, const char *fname, int count)
{
    // 以追加方式打开文件
    int fd = open(fname, O_CREAT | O_RDWR | O_APPEND, 0666);
    if (fd == -1)
    {
        cout << "文件打开失败!" << endl;
        exit(-1);
    }
    // 开始计时
    auto startTime = high_resolution_clock::now();

```

```

// 写入文件
for (int i = 0; i < count; i++)
    write(fd, str, strlen(str));
// 结束计时
auto endTime = high_resolution_clock::now();
cout << "writeTime:\t" << duration_cast<milliseconds>(endTime -
startTime).count() << " ms" << endl;
// 关闭文件
close(fd);
}

int main()
{
    const char *str = "Good Moring,FWM\n";
    const char *fname = "file_data";
    cout << "追加写文件 100 万次时间对比" << endl;
    // fwrite
    fwriteTime(str, fname, COUNT);
    // 清空文件
    truncate(fname, 0);
    // write
    writeTime(str, fname, COUNT);
    // 删除文件
    remove(fname);
    cout << "\nCopyright (c) 2023 by FWM-██████████, All Rights
Reserved.\n"
        << endl;
    return 0;
}

```

★ 运行截图

```

fwm-0100@ubuntu-18: ~/OS/实验2/源码
fwm-0100@ubuntu-18:~/OS/实验2/源码$ g++ FileMgmt_fw.cpp -o FileMgmt_fw
fwm-0100@ubuntu-18:~/OS/实验2/源码$ ./FileMgmt_fw
追加写文件100万次时间对比
fwriteTime:      71 ms
writeTime:      8359 ms

Copyright (c) 2023 by FWM-32116160100 , All Rights Reserved.

fwm-0100@ubuntu-18:~/OS/实验2/源码$

```



## 2) 模拟实现 ls -l 功能

### 源代码

```
/*
 * @Author: 
 * @Date: 2023-05-08 15:08:11
 * @LastEditTime: 2023-05-09 14:11:21
 * @FilePath: /源码/FileMgmt_ls.c
 * @Description: 模拟 ls -l 命令
 * Copyright (c) 2023 by GZHU-FWM, All Rights Reserved.
 */
#include <dirent.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>

#define MAXLEN 256 // 路径最大长度
// 终端文字颜色相关的宏
#define NONE "\033[m"
#define YELLOW "\033[1;33m"
#define LIGHT_BLUE "\033[1;34m"
#define LIGHT_CYAN "\033[1;36m"

// 获取当前路径
void getCurPath(char *path)
{
    // 获取当前目录路径
    if (getcwd(path, MAXLEN * sizeof(char)) != NULL)
        printf("%s\n\n", path);
    else
        printf("获取当前目录路径失败! \n");
}

// ls
void simList(const char *path)
{
    DIR *dir = NULL; // 目录指针
    struct dirent *dirent = NULL; // 目录项的结构体
    dir = opendir(path);
```



```

if (dir == NULL)
{
    printf("打开目录失败！检查目录是否存在\n");
    exit(-1);
}
while (dirent = readdir(dir))
{
    if (strcmp(".", dirent->d_name) && strcmp("..", dirent->d_name))
        printf("%s\n", dirent->d_name);
}
printf("\n");
closedir(dir);
}
// ls - l
void dtlList(const char *path)
{
    DIR *dir = NULL;           // 目录指针
    struct stat st;            // 文件信息结构图
    struct dirent *dirent = NULL; // 目录项的结构体
    char date[20];             // 日期字符串
    char temPath[MAXLEN];      // 用于拼接的相对路径
    int total = 0;             // 磁盘块数总和

    // 打开目录
    dir = opendir(path);

    // 开始逐项读取
    while (dirent = readdir(dir))
    {
        // 清空字符串
        memset(temPath, 0, sizeof(temPath));

        // 进行路径的拼接，采用绝对路径
        strcpy(temPath, path);
        strcat(temPath, "/");
        strcat(temPath, dirent->d_name);

        // 获取路径对于文件的信息
        lstat(temPath, &st);

        // --- 获取文件类型 ---
        if (S_ISREG(st.st_mode))
            printf("-");
        else if (S_ISDIR(st.st_mode))

```

```

        printf("d");
    else if (S_ISLNK(st.st_mode))
        printf("l");
    else if (S_ISCHR(st.st_mode))
        printf("c");
    else if (S_ISBLK(st.st_mode))
        printf("b");
    else if (S_ISFIFO(st.st_mode))
        printf("p");
    else if (S_ISSOCK(st.st_mode))
        printf("s");

    // --- 获取权限信息 ---
    // 所有者
    printf((st.st_mode & S_IRUSR) ? "r" : "-");
    printf((st.st_mode & S_IWUSR) ? "w" : "-");
    printf((st.st_mode & S_IXUSR) ? "x" : "-");
    // 所属组
    printf((st.st_mode & S_IRGRP) ? "r" : "-");
    printf((st.st_mode & S_IWGRP) ? "w" : "-");
    printf((st.st_mode & S_IXGRP) ? "x" : "-");
    // 其他用户
    printf((st.st_mode & S_IROTH) ? "r" : "-");
    printf((st.st_mode & S_IWOTH) ? "w" : "-");
    printf((st.st_mode & S_IXOTH) ? "x" : "-");
    printf(" ");

    // 硬链接数
    printf("%-*ld", 4, (long)st.st_nlink);
    // 文件所有者
    printf("%-*s ", 8, getpwuid(st.st_uid)->pw_name);
    // 文件所属组
    printf("%-*s ", 8, getgrgid(st.st_gid)->gr_name);
    // 文件大小
    printf("%ld\t", (long)st.st_size);
    // 最后被修改时间
    strftime(date, sizeof(date), "%Y-%m-%d %H:%M:%S",
        localtime(&st.st_mtime));
    printf("%s ", date);

    // --- 名称 ---
    // 可执行文件
    if ((st.st_mode & S_IXUSR) || (st.st_mode & S_IXGRP) || (st.st_mode
& S_IXOTH))

```

```

        if (strcmp(".", dirent->d_name) && strcmp("..",
dirent->d_name) && S_ISREG(st.st_mode))
            printf("%s%s*\n", LIGHT_CYAN, dirent->d_name, NONE);
        else
            printf("%s%s/\n", YELLOW, dirent->d_name, NONE);
        else
            printf("%s\n", dirent->d_name);

        // 记录文件磁盘块数
        total += st.st_blocks;
    }
    printf("%stotal: %d%s\n", LIGHT_BLUE, (total / 2), NONE);
    printf("\nCopyright (c) 2023 by FWM-██████████, All Rights
Reserved.\n\n");

    closedir(dir);
}

int main(int argc, char*argv[])
{
    char path[MAXLEN]; // 路径

    switch (argc)
    {
        // 没有输入任何参数
        case 1:
            getCurPath(path);
            simList(path);
            break;
        // 工作模式
        case 2:
            getCurPath(path);
            // ls -l
            if (!strcmp("-l", argv[1]))
                dtlList(path);
            else
            {
                printf("选项错误! \n");
                exit(-1);
            }
            break;
        // 工作模式 + 工作路径
        case 3:
            // 拷贝路径

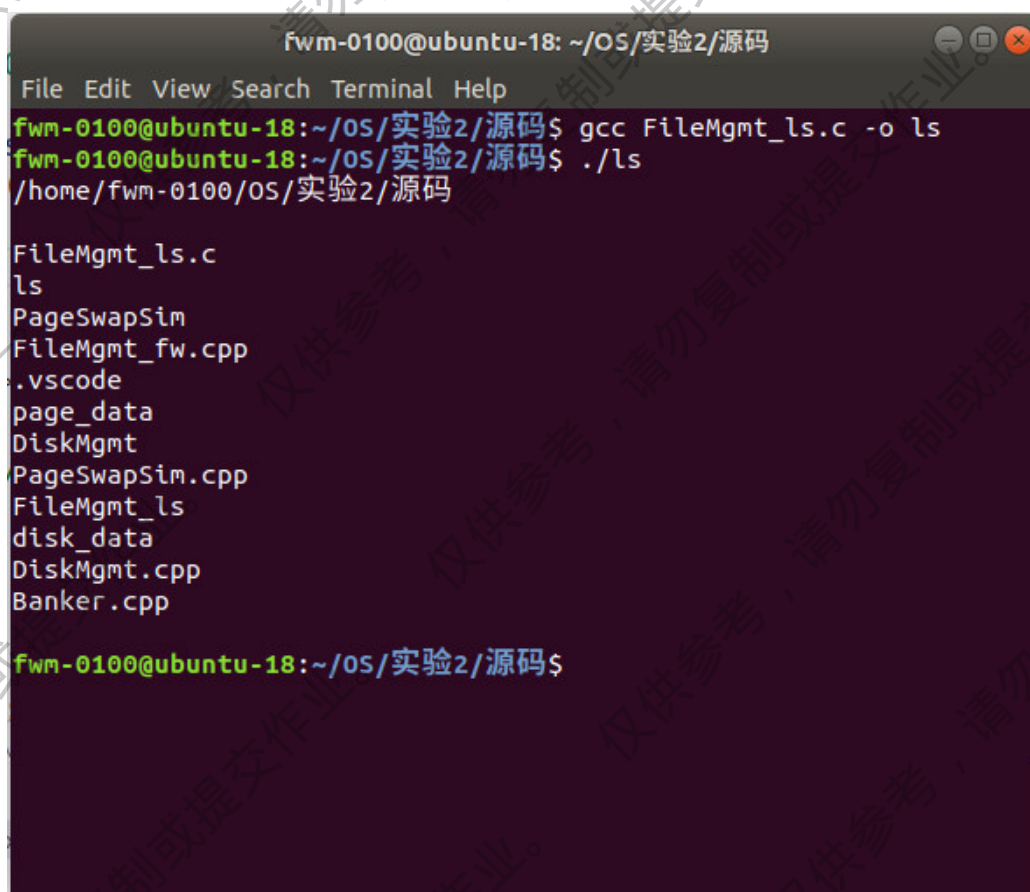
```

```

strcpy(path, argv[2]);
printf("%s\n\n", path);
// ls -l
if (!strcmp("-l", argv[1]))
    dtlList(path);
else
{
    printf("选项错误! \n");
    exit(-1);
}
break;
default:
    printf("参数过多! \n");
    exit(-1);
    break;
}
return 0;
}

```

运行截图（具体运行情况可见图注说明）



```

fwm-0100@ubuntu-18: ~/OS/实验2/源码
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验2/源码$ gcc FileMgmt_ls.c -o ls
fwm-0100@ubuntu-18:~/OS/实验2/源码$ ./ls
/home/fwm-0100/OS/实验2/源码

FileMgmt_ls.c
ls
PageSwapSim
FileMgmt_fw.cpp
.vscode
page_data
DiskMgmt
PageSwapSim.cpp
FileMgmt_ls
disk_data
DiskMgmt.cpp
Banker.cpp

fwm-0100@ubuntu-18:~/OS/实验2/源码$

```

图 1 没有任何参数，默认列出当前目录下的文件



```
fwm-0100@ubuntu-18: ~/OS/实验2/源码
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验2/源码$ gcc FileMgmt_ls.c -o ls
fwm-0100@ubuntu-18:~/OS/实验2/源码$ ./ls -l
/home/fwm-0100/OS/实验2/源码

-rw-rw-r-- 1 fwm-0100 fwm-0100 5246 2023-05-09 14:11:26 FileMgmt_ls.c
drwxrwxr-x 3 fwm-0100 fwm-0100 4096 2023-05-09 14:11:38 ./
-rwxrwxr-x 1 fwm-0100 fwm-0100 13328 2023-05-09 14:11:38 ls*
-rwxrwxr-x 1 fwm-0100 fwm-0100 30520 2023-05-03 10:29:39 PageSwapSim*
drwxr-xr-x 3 fwm-0100 fwm-0100 4096 2023-04-24 11:36:24 ../
-rw-rw-r-- 1 fwm-0100 fwm-0100 2141 2023-05-08 15:07:01 FileMgmt_fw.cpp
drwxrwxr-x 2 fwm-0100 fwm-0100 4096 2023-04-26 10:47:46 .vscode/
-rw-rw-r-- 1 fwm-0100 fwm-0100 106 2023-05-04 15:52:41 page_data
-rwxrwxr-x 1 fwm-0100 fwm-0100 166312 2023-05-07 17:01:40 DiskMgmt*
-rw-rw-r-- 1 fwm-0100 fwm-0100 20525 2023-05-03 10:30:49 PageSwapSim.cpp
-rwxrwxr-x 1 fwm-0100 fwm-0100 20272 2023-05-09 14:08:42 FileMgmt_ls*
-rwxr-w-r-- 1 fwm-0100 fwm-0100 32 2023-05-04 15:52:45 disk_data*
-rw-rw-r-- 1 fwm-0100 fwm-0100 11934 2023-05-07 17:01:37 DiskMgmt.cpp
-rw----- 1 fwm-0100 fwm-0100 7338 2023-05-07 09:49:42 Banker.cpp
total: 308

Copyright (c) 2023 by FWM-32116160100 , All Rights Reserved.

fwm-0100@ubuntu-18:~/OS/实验2/源码$
```

图 2 参数-l, 执行与系统命令 ls -l 相同功能 (有目录颜色标注、可执行文件的\*标志)

```
fwm-0100@ubuntu-18: ~/OS/实验2/源码
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验2/源码$ gcc FileMgmt_ls.c -o ls
fwm-0100@ubuntu-18:~/OS/实验2/源码$ ./ls -l ~/OS
/home/fwm-0100/OS

drwxr-xr-x 2 fwm-0100 fwm-0100 4096 2023-04-24 11:07:02 源/
drwxr-xr-x 3 fwm-0100 fwm-0100 4096 2023-04-24 11:36:24 实验2/
drwxrwxr-x 8 fwm-0100 fwm-0100 4096 2023-05-04 18:45:32 ./
drwxr-xr-x 25 fwm-0100 fwm-0100 4096 2023-05-08 19:31:59 ../
drwxr-xr-x 5 fwm-0100 fwm-0100 4096 2023-05-04 15:42:50 实验1/
drwxr-xr-x 4 root root 4096 2023-05-04 18:46:11 bochs/
drwxr-xr-x 2 fwm-0100 fwm-0100 4096 2023-04-23 10:16:46 实验0/
-rw-rw-rw- 1 fwm-0100 fwm-0100 593898 2023-04-24 11:25:45 操作系统实验指导书 (实验一、二).pdf
drwxr-xr-x 4 fwm-0100 fwm-0100 4096 2023-05-04 18:50:03 实验3/
total: 612

Copyright (c) 2023 by FWM-32116160100 , All Rights Reserved.

fwm-0100@ubuntu-18:~/OS/实验2/源码$
```

图 3 参数-l + <path>, 列出指定路径下的文件

#### 4. 文件管理实验

- 从文件中读取数据
- 模拟先来先服务法 (First-Come, First-Served, FCFS), 最短寻道时间优先法 (Shortest Seek Time First, SSTF), 电梯法三种磁盘调度算法, 输出为每种调度算法的磁头移动轨迹和移动的总磁道数。

##### ★ 源代码

```
/*
 * @Author: 
 * @Date: 2023-05-04 16:31:46
 * @LastEditTime: 2023-05-07 17:01:37
 * @FilePath: /源码/DiskMgmt.cpp
```

```

* @Description: 磁盘管理实验
* Copyright (c) 2023 by GZHU-FWM, All Rights Reserved.
*/

#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include <fstream>
#include <iomanip>
#include <limits.h>

using namespace std;

// 终端文字颜色相关的宏
#define NONE "\033[m"
#define YELLOW "\033[1;33m"
#define LIGHT_BLUE "\033[1;34m"
#define LIGHT_CYAN "\033[1;36m"

// ----- 快速排序 -----
int Partition(vector<int> &A, int low, int high); // 划分
void QuickSort(vector<int> &A, int low, int high); // 快排

// ----- 测试数据 -----
class DiskData
{
public:
    void InitData(const string &fname); // 从文件中读取数据初始化数据成员
    void PrintData(); // 打印初始化后的数据

    int start; // 起始磁道号
    vector<int> disklist; // 磁道号序列
    vector<int> sortlist; // 排序后的磁道号序列
} dd;

void FCFS(); // 先来先服务算法 FCFS

int SSTF_FindFirst(const vector<int> &sortlist, int
start); // 找到第一个服务的磁道号下标
int SSTF_FindNext(const vector<int> &sortlist, vector<bool> &flag, int
head); // 用于找到下一个服务的磁道号下标

```



```

void
SSTF();
// 最短寻道时间优先算法 SSTF

int LOOK_FindFirst(const vector<int> &sortlist, int start, int
dir); // 找到第一个服务的磁道号下标
void LOOK_Move(const vector<int> &sortlist, int head, int start, int dir);
// 电梯移动
void LOOK(); // 电梯算法 LOOK

int main(int argc, char *argv[])
{
    // 读取参数
    if (argc != 2)
    {
        cout << "ERROR: 参数输入错误! 请检查后重试。" << endl;
        exit(-1);
    }
    string fname = argv[1];
    cout << "输入参数为: " << fname << endl;

    dd.InitData(fname);
    dd.PrintData();

    // 开始模拟各种磁道算法
    FCFS();
    SSTF();
    LOOK();
    return 0;
}

// ----- 快速排序 -----
// 划分
int Partition(vector<int> &A, int low, int high)
{
    // 选取下标为 low 的元素作为划分枢纽
    int pivot = A[low];
    // 寻找枢纽的位置下标
    while (low < high)
    {
        // 在 high 部分找到一个不属于 high 的元素
        while (low < high && A[high] >= pivot)
            high--;
        // 然后将其替换到 low 部分去

```

---

```

    A[low] = A[high];
    // 同理，在 low 部分找到不属于的元素
    while (low < high && A[low] <= pivot)
        low++;
    // 替换到 high 部分去
    A[high] = A[low];
}
// 枢纽放入分界位置
A[low] = pivot;
// 返回
return low;
}
// 快排
void QuickSort(vector<int> &A, int low, int high)
{
    // 递归终止条件
    if (low < high)
    {
        // 找到一个分界位置
        int pivot = Partition(A, low, high);
        // 递归分界左边
        QuickSort(A, low, pivot - 1);
        // 递归分界右边
        QuickSort(A, pivot + 1, high);
    }
}

// ----- 测试数据 -----
void DiskData::InitData(const string &fname)
{
    string line;
    ifstream ifs;
    stringstream ss;

    cout << "\nCopyright (c) 2023 by FWM-██████████, All Rights
    Reserved.\n"
         << endl;

    // 打开文件
    ifs.open(fname);
    if (!ifs.is_open())
    {
        cout << "文件打开失败!" << endl;
        return;
    }
}

```

```

}

// 读取文件
// i 为行号
for (int i = 1; getline(ifs, line); i++)
{
    switch (i)
    {
        case 1:
            // 读取起始磁道号
            ss.str(line);
            ss >> start;
            ss.clear();
            break;

        case 2:
            int num = -1; // 存储每次从 ss 里读取的数
            // 初始化 ss
            ss.str(line);
            while (ss >> num)
                disklist.push_back(num);
            break;
    }
}

// 关闭文件
ifs.close();
// 排序
sortlist = disklist;
QuickSort(sortlist, 0, sortlist.size() - 1);
}

void DiskData::PrintData()
{
    cout << "文件内容: \n";
    // 起始磁道号
    cout << start << endl;
    // 磁道号序列
    for (int i = 0; i < disklist.size(); i++)
        cout << disklist[i] << " ";
    cout << endl;
}

// ----- 先来先服务算法 FCFS -----
void FCFS()

```

```

{
    int head = dd.start; // head 模拟的磁头, head 是磁道号
    int count = 0;      // 磁头移动的总距离

    cout << YELLOW << "先来先服务法(FCFS)" << endl;
    cout << NONE << "磁头服务顺序: ";
    cout << head << " ";
    // FCFS, 直接遍历即可
    for (int i = 0; i < dd.disklist.size(); i++)
    {
        cout << dd.disklist[i] << ' ';
        count += abs(dd.disklist[i] - head);
        head = dd.disklist[i];
    }
    cout << "\n 磁头移动的总磁道数为: " << LIGHT_CYAN << count << endl;
    cout << NONE << "平均寻找长度为: " << (double)count /
(double)dd.disklist.size() << endl;
}

// ----- 最短寻道时间优先算法 SSTF -----
// 找到第一个服务的磁道号下标
int SSTF_FindFirst(const vector<int> &sortlist, int start)
{
    int i = 0;
    int low = 0, high = sortlist.size() - 1, mid = -1;
    // 当 start 小于等于 A[0] 时特殊处理
    if (start <= sortlist[0])
        return 0;
    // 当 start 大于等于 A[high] 时特殊处理
    if (start >= sortlist[high])
        return high;
    // 二分查找
    while (low < high)
    {
        mid = (low + high) / 2;
        if (start >= sortlist[mid])
            low = mid + 1;
        else
            high = mid;
    }

    // 比较位置 high 与 high-1 位置哪个更靠近 start (high 位置是恰好>start;
    // high-1 位置恰好<start)
    int dist1 = abs(sortlist[high] - start);

```

```

int dist2 = abs(sortlist[high - 1] - start);
if (dist1 <= dist2)
    return high;
else
    return high - 1;
return -1;
}
// 用于找到下一个服务的磁道号下标
int SSTF_FindNext(const vector<int> &sortlist, vector<bool> &flag, int
head)
{
    // 往左看
    int frontPos = head - 1;
    int frontRec = INT_MAX;
    // 找到第一个没被访问过的
    while (frontPos >= 0)
    {
        if (!flag[frontPos])
            break;
        frontPos--;
    }
    // 计算与 sortlist[head]的差
    if (frontPos != -1)
        frontRec = abs(sortlist[frontPos] - sortlist[head]);

    // 往右看
    int rearPos = head + 1;
    int rearRec = INT_MAX;
    // 找到第一个没被访问过的
    while (rearPos < sortlist.size())
    {
        if (!flag[rearPos])
            break;
        rearPos++;
    }
    // 计算与 sortlist[head]的差
    if (rearPos != sortlist.size())
        rearRec = abs(sortlist[rearPos] - sortlist[head]);

    // 比较左右, 返回距离 data[head]最近的
    // 左右距离相等
    if (frontRec == rearRec && frontRec != INT_MAX)
        return frontPos;
    // 右边更近

```

```

else if (frontRec > rearRec)
    return rearPos;
// 左边更近
else if (frontRec < rearRec)
    return frontPos;
else
    return -1;
}
// 最短寻道时间优先算法 SSTF
void SSTF()
{
    int count = 0; // 磁头移动的总距离
    int head = 0; // head 模拟的磁头, head 是下标!!
    int qhead = 0; // 记录 head 的上一个位置
    vector<bool> flag(dd.disklist.size(), false); // 标记数组, 是否已经访问

    cout << YELLOW << "\n 最短寻道时间优先算法(SSTF)\n";
    cout << NONE << "磁头服务顺序: ";
    cout << dd.start << " ";

    // 找到磁头第一个服务的磁道号下标
    head = SSTF_FindFirst(dd.sortlist, dd.start);
    flag[head] = true;
    // 记录磁头移动距离
    count += abs(dd.start - dd.sortlist[head]);
    cout << dd.sortlist[head] << " ";
    // 记录磁头上一个位置 (便于计算移动距离)
    qhead = head;
    while ((head = SSTF_FindNext(dd.sortlist, flag, head)) != -1)
    {
        flag[head] = true;
        count += abs(dd.sortlist[head] - dd.sortlist[qhead]);
        cout << dd.sortlist[head] << " ";
        qhead = head;
    }
    cout << endl;
    cout << "磁头移动的总磁道数为: " << LIGHT_CYAN << count << endl;
    cout << NONE << "平均寻找长度为: " << (double)count /
(double)dd.sortlist.size() << endl;
}

// ----- 电梯算法 LOOK -----
// 找到第一个服务的磁道号下标

```



```

int LOOK_FindFirst(const vector<int> &sortlist, int start, int dir)
{
    int i = 0;
    int low = 0, high = sortlist.size() - 1, mid = -1;
    // 当 start 小于等于 A[0] 时特殊处理
    if (start <= sortlist[0])
        return 0;
    // 当 start 大于等于 A[high] 时特殊处理
    if (start >= sortlist[high])
        return high;
    // 二分查找
    while (low < high)
    {
        mid = (low + high) / 2;
        if (start >= sortlist[mid])
            low = mid + 1;
        else
            high = mid;
    }
    // high 是恰好大于 start 的下标, high-1 是恰好小于 start 的下标
    // 根据方向返回下标
    // 向磁道号减小的方向移动
    if (dir == 0)
        return high - 1;
    // 向磁道号增大的方向移动
    else
        return high;
}

// 电梯移动
void LOOK_Move(const vector<int> &sortlist, int head, int start, int dir)
{
    // 电梯扫描, 磁道号增大、减小方向各一次
    for (int i = 0; i < 2; i++)
    {
        // 减小方向移动
        if (dir == 0)
        {
            // 不断左移
            while (head >= 0)
            {
                cout << sortlist[head] << " ";
                head--;
            }
            // 指针左移到最小磁道号, 更新指针到开始位置右侧
        }
    }
}

```

```

        head = start + 1;
        // 设置移动的方向(0 左 1 右)
        dir = 1;
    }
    // 增大方向移动
    else
    {
        // 同理
        while (head < sortlist.size())
        {
            cout << sortlist[head] << " ";
            head++;
        }
        head = start - 1;
        dir = 0;
    }
}
// 电梯算法 LOOK
void LOOK()
{
    int head = 0; // head 模拟的磁头, head 是下标!!
    int qhead = 0; // 记录 head 的上一个位置
    int count = 0; // 磁头移动的总距离

    cout << YELLOW << "\n 电梯算法(LOOK)\n";

    // ----- 磁头向增大方向移动 -----
    // 找到距离起点最近的磁道
    head = LOOK_FindFirst(dd.sortlist, dd.start, 1);
    // 记录 head 此刻的位置
    qhead = head;
    cout << LIGHT_BLUE << "1.磁头方向: 磁道号增加\n";
    cout << NONE << "磁头服务顺序: ";
    cout << dd.start << " ";
    LOOK_Move(dd.sortlist, head, head, 1);
    cout << endl;
    count = (dd.sortlist.back() - dd.start) + (dd.sortlist.back() -
dd.sortlist.front());
    cout << "磁头移动的总磁道数为: " << LIGHT_CYAN << count << endl;
    cout << NONE << "平均寻找长度为: " << (double)count /
(double)dd.sortlist.size() << endl;

    // ----- 磁头向减小方向移动 -----

```

```

// 找到距离起点最近的点
head = LOOK_FindFirst(dd.sortlist, dd.start, 0);
cout << LIGHT_BLUE << "2.磁头方向: 磁道号减少\n";
cout << NONE << "磁头服务顺序: ";
cout << dd.start << " ";
LOOK_Move(dd.sortlist, head, head, 0);
cout << endl;
count = (dd.start - dd.sortlist.front()) + (dd.sortlist.back() -
dd.sortlist.front());
cout << "磁头移动的总磁道数为: " << LIGHT_CYAN << count << endl;
cout << NONE << "平均寻找长度为: " << (double)count /
(double)dd.sortlist.size() << endl;
}

```

★ 运行截图

```

fwm-0100@ubuntu-18: ~/OS/实验2/源码
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验2/源码$ ./DiskMgmt disk_data
输入参数为: disk_data

Copyright (c) 2023 by FWM-32116160100 , All Rights Reserved.

文件内容:
53
98 183 37 122 14 124 65 67

先来先服务法(FCFS)
磁头服务顺序: 53 98 183 37 122 14 124 65 67
磁头移动的总磁道数为: 640
平均寻找长度为: 80

最短寻道时间优先算法(SSTF)
磁头服务顺序: 53 65 67 37 14 98 122 124 183
磁头移动的总磁道数为: 236
平均寻找长度为: 29.5

电梯算法(LOOK)
1. 磁头方向: 磁道号增加
磁头服务顺序: 53 65 67 98 122 124 183 37 14
磁头移动的总磁道数为: 299
平均寻找长度为: 37.375
2. 磁头方向: 磁道号减少
磁头服务顺序: 53 37 14 65 67 98 122 124 183
磁头移动的总磁道数为: 208
平均寻找长度为: 26
fwm-0100@ubuntu-18:~/OS/实验2/源码$

```

## 四、实验分析与思考

### 1. 银行家算法

银行家算法是一种通过安全性检查来避免死锁的算法,用于管理多个进程之间共享有限数量资源的分配。该算法可以确保系统分配资源时不会出现死锁现象。银行家算法需要知道每个进程的最大资源需求量、已经获得的资源数量以及当前请求的资源数量,才能进行资源分配和判断。银行家算法会检查每次资源请求的合法性,只有当请求合法时才会进行资源分配,否则就会等待直到请求变得合法为止。

在我的实验程序中实现从文件读取  $t_0$  时刻资源的功能,大大减少了输入的数据量。

### 2. 常用页面置换算法模拟

在实验中实现了 OPT、FIFO、LRU 三种算法。

OPT: 选择在未来最长时间不会被使用的页面进行淘汰。可以理论上得到最优的页面置换方案,缺页率最低。但其需要预知未来,无法实现,故仅可以作为“标尺”来衡量页面置换算法的优劣性。

FIFO: 选择最先进入内存的页面进行淘汰。其实现简单,但是会出现 Belady 异常。

LRU: 选择最近最少使用的页面进行淘汰。避免了 Belady 异常,缺页率表现比 FIFO 要更好。LRU 算法是对“过去”的总结,通过过去的经验来预测未来的页面访问序列。但是在当内存容量较小时,LRU 算法可能出现大量缺页的情况 Thrashing 即抖动。

我尝试在不同数量的数据块下测试上述三种算法,并且作出不同数量内存块 (3-9) 下三种算法的命中率曲线图 (使用 `gunplot`), 使用指导书给的测试文件测试结果如下。可以看出 FIFO 的曲线非常曲折, LRU 的曲线较光滑相对接近最光滑的 OPT 曲线。在命中率方面可以看出  $OPT > FIFO > LRU$ 。

程序代码如下 (在实验源程序基础上更改,完整代码可见 `PageSwapSim_block.cpp`):

```
// ----- 测试不同数量内存块下的页面置换算法 -----
void DiffBlockNum(int min, int max)
{
    // 绘图
    FILE *gnuplot = popen("gnuplot -persist", "w");
    vector<vector<double>> result(max - min + 1, vector<double>(TYPENUM,
-1));
    cout << endl;
    for (int blockNum = min; blockNum <= max; blockNum++)
    {
        // 设置内存块数量
```

```

pd.SetBlockNum(blockNum);
// 运行算法 记录结果
result[blockNum - min][0] = OPT();
result[blockNum - min][1] = FIFO();
result[blockNum - min][2] = LRU();
// 输出结果, 小数点后保留两位
cout << "blockNum: " << blockNum << endl
    << result[blockNum - min][0] << "% "
    << result[blockNum - min][1] << "% "
    << result[blockNum - min][2] << "%" << endl
    << endl;

// 将上面的命中率结果绘制成曲线图
fprintf(gnuplot, "set xlabel '用户内存容量'\n");
fprintf(gnuplot, "set ylabel '命中率'\n");
fprintf(gnuplot, "set xrange [3:9]\n");
fprintf(gnuplot, "set yrange [45:85]\n");
fprintf(gnuplot, "plot '-' with lines linewidth 2 title 'OPT', ");
fprintf(gnuplot, "'-' with lines linewidth 2 linecolor rgb 'black'
title 'FIFO', ");
fprintf(gnuplot, "'-' with lines linewidth 2 title 'LRU'\n");
for (int j = 0; j < max - min + 1; j++)
    fprintf(gnuplot, "%d %lf\n", j + min, result[j][0]);
fprintf(gnuplot, "e\n");
for (int j = 0; j < max - min + 1; j++)
    fprintf(gnuplot, "%d %lf\n", j + min, result[j][1]);
fprintf(gnuplot, "e\n");
for (int j = 0; j < max - min + 1; j++)
    fprintf(gnuplot, "%d %lf\n", j + min, result[j][2]);
fprintf(gnuplot, "e\n");
fflush(gnuplot);
}
}

int main(int argc, char *argv[])
{
    // 读取参数
    if (argc != 2)
    {
        cout << "ERROR: 参数输入错误! 请检查后重试。" << endl;
        exit(-1);
    }
    string fname = argv[1];
    cout << "输入参数为: " << fname << endl;
}

```



```

// 初始化
pd.InitData(fname);
pd.PrintData();

DiffBlockNum(3, 9);

exit(0);
}

```

```

fwm-0100@ubuntu-18: ~/OS/实验2/源码
fwm-0100@ubuntu-18:~/OS/实验2/源码$ ./PageSwapSim_block page_data
输入参数为：page_data

Copyright (c) 2023 by FWM-32116160100 , All Rights Reserved.

文件内容：
4
50
7 0 9 1 2 0 3 0 4 2 3 0 3 2 1 2 0 8 1 7 0 1 7 0 4 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 8 1 7 0 1 1 2 3 4 5

blockNum: 3
56% 24% 30%

blockNum: 4
64% 50% 58%

blockNum: 5
72% 56% 66%

blockNum: 6
78% 56% 70%

blockNum: 7
82% 78% 80%

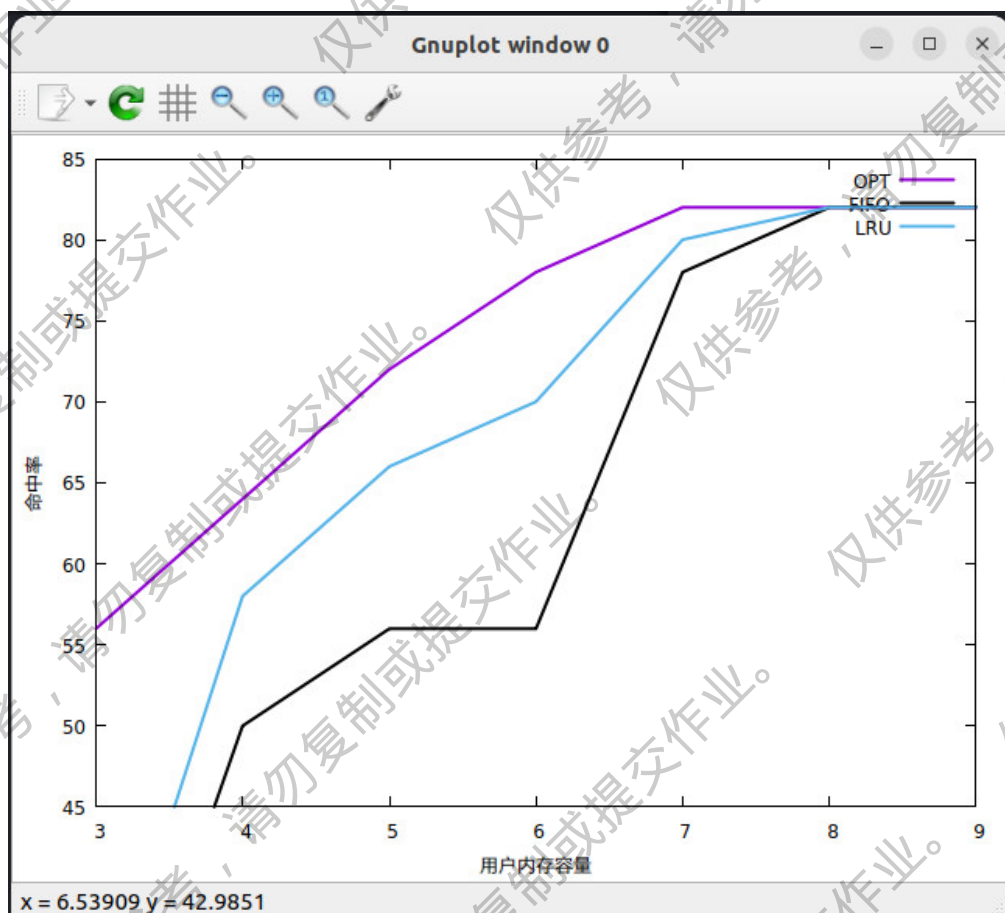
blockNum: 8
82% 82% 82%

blockNum: 9
82% 82% 82%

fwm-0100@ubuntu-18:~/OS/实验2/源码$ Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_
PLATFORM=wayland to run on Wayland anyway.
fwm-0100@ubuntu-18:~/OS/实验2/源码$

```





### 3. 文件管理

#### 对 fwrite 与 write 性能差距进行对比分析

在不断追加写文件 100 万次，内容为“Good Moring,FWM”的实验中，fwrite 需要的时间比 write 要短得多。

```

fwriteTime(str, name, count);
问题 输出 调试控制台 终端
追加写文件100万次时间对比
fwriteTime: 24 ms
writeTime: 1226 ms
Copyright (c) 2023 by FWM-32116160100 , All Rights Reserved.
[1] + Done "/usr/bin/gdb" --interpreter=mi --tty=$
fwm-0100@ubuntu-18:~/OS/实验2/源码$

```

fwrite 是一个 C 标准库接口，而 write 是一个系统内核提供的系统调用接口。两者所处的层次不一样，fwrite 是 write 的进一步抽象。

如下图所示，字符串 str 相当于应用层的 application buffer，当调用 fwrite(str, sizeof(char), strlen(str), fp)后，数据从 application buffer 拷贝到了 C 标准库内部管理的缓冲区 clib buffer，直到 clib buffer 满或者人为调用 fflush() / fclose()后才会刷新缓冲区，通过系统调用将数据写入到系统内核的 page cache 中，等待内核写回硬盘（如果不想等

待可以使用 fsync 函数刷新 page cache)。利用 fwrite 接口时, 需要经历 application buffer  
-> clib buffer -> page cache 三个过程。

与 fwrite 不同, write 是系统调用接口, 它可以直接一步到位: application buffer -> page cache。理论上应该是 write 的速度更加快, 因为其少了 clib buffer 的中间层。但实际上使用系统调用时需要进行上下文的切换等操作开销较大, write 写入 100 万次的字符串就需要进行 100 万次的系统调用, 100 万次系统开销巨大严重增加了写文件的时间。反而 fwrite 因为具有 clib buffer C 库缓存区, 可以累积数据, 一次系统调用写入更多数据, 减少了系统的调用的次数, 使每次系统调用的效率更高, 完成 100 万次写入的任务的开销更小。

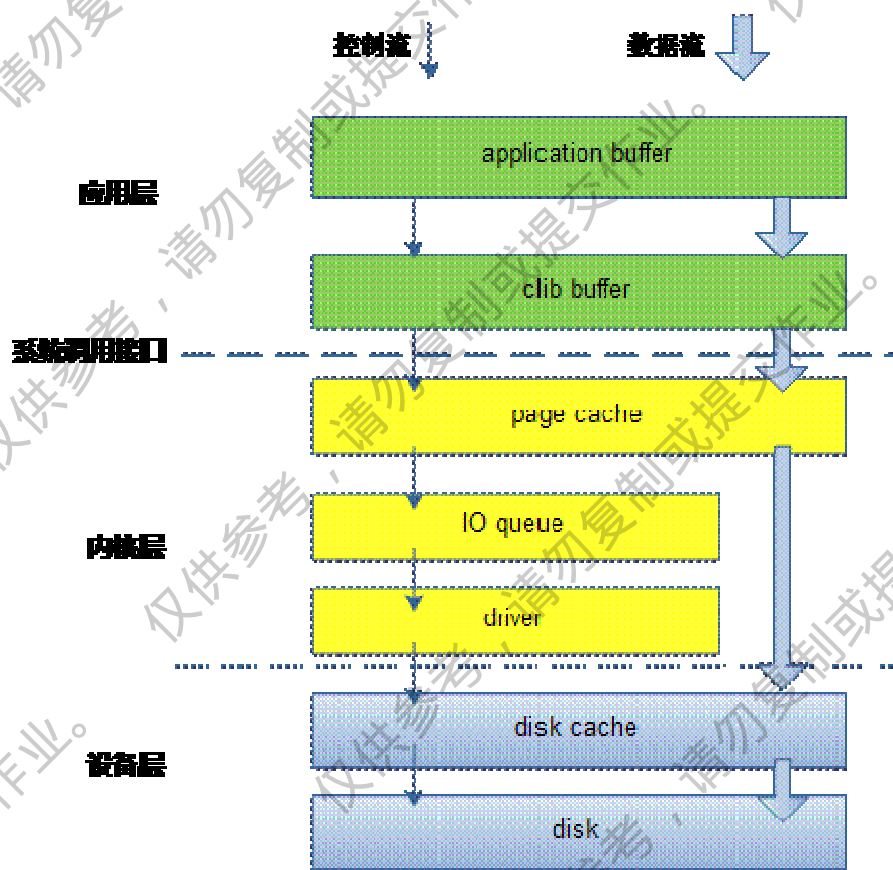


图 4 文件 IO 分层图

(源: <http://blog.chinaunix.net/uid-27105712-id-3270102.html>)

#### 4. 磁盘管理

本实验实现了 FCFS、SSTF、SCAN 电梯三种磁盘调度算法。

其中 FCFS 实现非常简单粗暴，不需要任何另外的数据结构，但是性能非常差且影响磁头寿命。SSTF 与 SCAN 实现都需要先对磁道访问序列进行排序，SSTF 需要找到距离当前磁头位置最靠近的两个磁道，需要在排序后的磁道访问序列向前、后找距离最近、未被访问的磁道号，时间复杂度达到  $O(n^2)$ ，还可能出现饥饿现象；SCAN 实现较为简单只要找到磁头起始位置下一个访问的磁道号即可，时间复杂度为  $O(n \log n)$ 。