内存管理实验

班级：2022211304 学号：20222211119 姓名：赵宇鹏

1. 实验目的

基于 openEuler 操作系统，通过模拟实现按需调页式存储管理的几种基本页面置换算法，了解虚拟存储技术的特点，掌握虚拟存储按需调页式存储管理中几种基本页面置换算法的基本思想和实现过程，并比较它们的效率。

2. 实验内容及要求

2.1 生成内存访问串

首先用 srand()和 rand()函数定义和产生指令地址序列，然后将指令地址序列变换成相应的页地址流。

比如：通过随机数产生一个内存地址，共 100 个地址，地址按下述原则生成 :

（1）70%的指令是顺序执行的

（2）10%的指令是均匀分布在前地址部分

（3）20%的指令是均匀分布在后地址部分

具体的实施方法是:

（a）从地址 0 开始;

（b）若当前指令地址为 m，按上面的概率确定要执行的下一条指令地址，分别为顺序、在前和在后:

顺序执行 : 地址为 m+1 的指令;

在前地址 : [0，m-1]中依前面说明的概率随机选取地址;

在后地址 : [m+1，99]中依前面说明的概率随机选取地址;

（c）重复（b）直至生成 100 个指令地址。

假设每个页面可以存放 10（可以自己定义）条指令，将指令地址映射到页面，生成内存访问串。

2.2 设计算法，计算访问缺页率并对算法的性能加以比较

（1）最优置换算法（Optimal）

（2）最近最少使用（Least Recently Used）

（3）先进先出法（Fisrt In First Out）

其中，缺页率= 页面失效次数/ 页地址流长度

要求 :

分析在同样的内存访问串上执行，分配的物理内存块数量和缺页率之间的关系; 并在同样情况下，对不同置换算法的缺页率比较。

3. 实验环境及主要API说明

3.1 实验环境

1. WSL 2.2.4.0

Windows Subsystem for Linux (WSL) 是一个允许用户在 Windows 操作系统 上运行 Linux 二进制可执行文件的兼容层。通过 WSL，用户无需使用虚拟机或双 系统，即可在 Windows 中运行大多数 Linux 命令行工具和应用程序。

2. vscode

3. gcc version 13.2.0

3.2 API 介绍

本实验主要调用了`srand`和`rand`函数，此外还有`generate_memory_access_sequence`函数，`map_to_page_sequence`函数，`FIFO`函数，`LRU`函数，`OPT`函数等模块，接下来进行逐一说明：

1. `srand` **函数**

功能:
`void srand(unsigned int seed)`

- 用于设置随机数生成的种子，使得`rand`函数生成的随机序列可控。
- 如果种子值相同，每次生成的随机序列也相同；如果种子值不同，生成的序列会不同。

2. `rand` **函数**

功能:
`int rand(void)`

- 用于生成伪随机数，返回范围是`[0, RAND_MAX]`，其中`RAND_MAX`是一个常量（通常为32767）。

3. `generate_memory_access_sequence(int *sequence)`

功能:
生成一个内存访问地址序列，模拟指令访问过程。

参数:

- `int *sequence`：用于存储生成的地址序列，长度为 100（即宏`ADDRESS_COUNT`）。

4. `map_to_page_sequence(int *address_sequence, int *page_sequence)`

功能:
将指令地址序列映射到对应的页地址序列。

参数:

- `int *address_sequence`：输入的指令地址序列。
- `int *page_sequence`：输出的页地址序列。

5. `FIFO(int *page_sequence, int block_num)`

功能:
基于**先进先出算法（FIFO）**计算缺页次数。

参数:

- `int *page_sequence`：输入的页地址序列。
- `int block_num`：物理内存块数量。

返回值：

- 缺页次数。

6. `LRU(int *page_sequence, int block_num)`

功能：
基于**最近最少使用算法（LRU）**计算缺页次数。

参数：

- `int *page_sequence`：输入的页地址序列。
- `int block_num`：物理内存块数量。

返回值：

- 缺页次数。

7. `OPT(int *page_sequence, int block_num)`

功能：
基于**最佳置换算法（OPT）**计算缺页次数。

参数：

- `int *page_sequence`：输入的页地址序列。
- `int block_num`：物理内存块数量。

返回值：

- 缺页次数。

4. 程序设计说明

4.1 随机数生成

随机数生成函数如下：

```
void generate_memory_access_sequence(int *sequence) {
    srand(time(NULL));
    int current_address = 0;
    sequence[0] = current_address;

    for (int i = 1; i < ADDRESS_COUNT; i++) {
        int rand_prob = rand() % 100;

        if (rand_prob < 70) {
            current_address = (current_address + 1) % ADDRESS_COUNT;
        } else if (rand_prob < 80) {
            if (current_address > 0) {
                current_address = rand() % current_address;
            } else {
                current_address = 0;
```

```
            }
        } else {
            if (current_address < ADDRESS_COUNT - 1) {
                current_address = current_address + 1 + rand() %
(ADDRESS_COUNT - current_address - 1);
            } else {
                current_address = ADDRESS_COUNT - 1;
            }
        }

        sequence[i] = current_address;
    }
}
```

思路与题目所给思路相同，在这里就不再做解释说明，然后使用页面映射函数，把指令地址映射到页面中，在这里，每个页面可以存储10条指令，代码如下：

```
void map_to_page_sequence(int *address_sequence, int *page_sequence) {
    for (int i = 0; i < ADDRESS_COUNT; i++) {
        page_sequence[i] = address_sequence[i] / PAGE_SIZE;
    }
}
```

4.2 FIFO算法

先进先出调度算法的实现比较简单，只需要设置一个`current_index`指针，当分配的内存块中的页面命中时，无需`pge in`操作，当没命中时，则需要进行`page in`操作，将需要的页面移入内存块中，`current_index`每次都指向下一次`page in`操作时应该被覆盖的页表，然后每次`page in`后向后移动一位，移动到最后则返回内存表开头，实现一个循环数组的作用，这样就实现了FIFO调度算法，代码如下：

```
int FIFO(int* page_sequence, int block_num) {
    int page_talbe[block_num];
    int page_fault = 0;
    int current_index = 0;
    //初始化
    for(int i = 0; i < block_num; i++) {
        page_talbe[i] = -1;
    }
    for(int i = 0; i < ADDRESS_COUNT; i++) {
        int hit = 0;
        for(int j = 0; j < block_num; j++) {
            if(page_talbe[j] == page_sequence[i]) {
                hit = 1;
                break;
            }
        }
        //未命中
        if(hit == 0) {
            page_fault++;
            page_talbe[current_index] = page_sequence[i];
            current_index = (current_index + 1) % block_num;
        }
```

```
    }
    return page_fault;
}
```

4.3 LRU算法

LRU算法在每次缺页时，`page in`写入的所需页面所覆盖的是最久没有使用的那一个，所以我使用`time_stamp[block_num]`数组来分别记录内存块中所储存的页面离上一次被调用的时间长短，具体实现是每一次执行一条指令时，对被调用的页面或者写入的页面的`time_stamp`，将其置0，表示刚被使用，而其他已经被赋值过的`time_stamp`则全都加一，然后使用`current_index`指针指向`time_stamp`最大的那一个内存块，这样当需要`page in`写入时，就将页面写入`current_index`所指向的那个内存块即可，代码实现如下：

```
int LRU(int* page_sequence, int block_num) {
    int page_talbe[block_num];
    int time_stamp[block_num];
    int page_fault = 0;
    //初始化
    for(int i = 0; i < block_num; i++) {
        page_talbe[i] = -1;
        time_stamp[i] = 0;
    }
    for(int i = 0; i < ADDRESS_COUNT; i++) {
        int hit = 0;
        int current_index = 0;
        int max_time = time_stamp[0];
        for(int j = 0; j < block_num; j++) {
            //内存块为空，无需覆盖直接写入
            if(page_talbe[j] == -1) {
                current_index = j;
                break;
            }
            //命中
            else if(page_talbe[j] == page_sequence[i]) {
                hit = 1;
                current_index = j;
                break;
            }
            //若既没命中，内存块也非空，则寻找最久没有调用的页面
            if(time_stamp[j] > max_time) {
                max_time = time_stamp[j];
                current_index = j;
            }
        }
        //增加各页面被调用的时间
        for(int j = 0; j < block_num; j++) {
            if(j == current_index) {
                time_stamp[j] = 0;
            }
            else {
                if(page_talbe[j] != -1) {
                    time_stamp[j]++;
                }
            }
```

```
        }
        //未命中，page in
        if(hit == 0) {
            page_fault++;
            page_talbe[current_index] = page_sequence[i];
        }
    }
    return page_fault;
}
```

4.4 Optimal算法

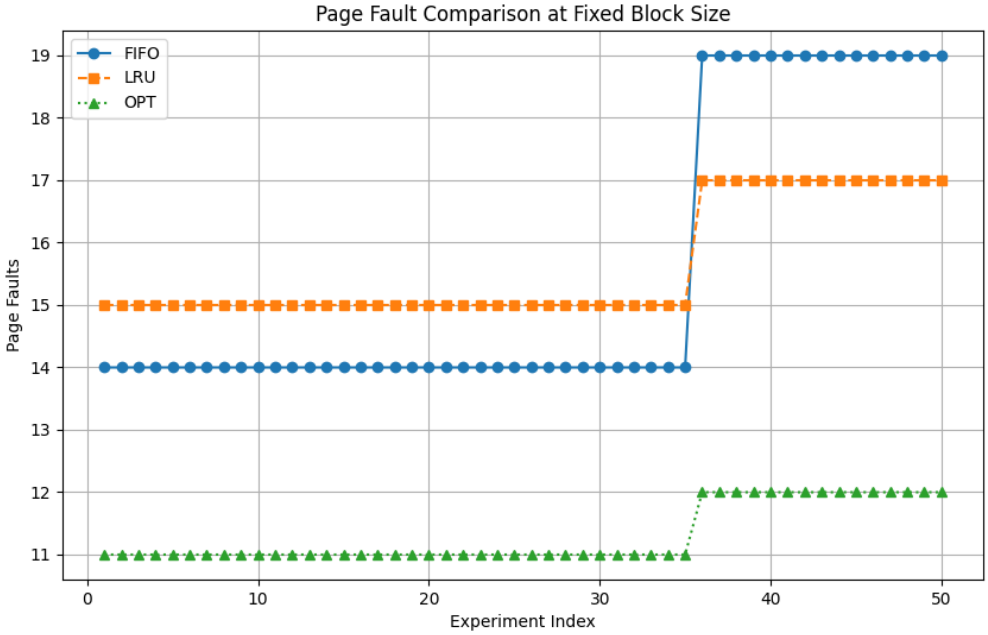
最优置换算法的思路是若内存块为空，则直接写入，若页面命中，则不管，若缺页，则需要进行页面置换，被置换的页面选取与`LRU算法`不同，通过对当前执行指令的后续指令进行查找和比较，找到找到当前内存块中存储的页面中最后被调用，或者没有被调用的页面，将其与当前所需的页面进行置换，这个方法比较暴力，但是对于数据量较小的情况还是可以使用的，代码实现如下：

```
int OPT(int* page_sequence, int block_num) {
    int page_talbe[block_num];
    int page_fault = 0;
    //初始化
    for(int i = 0; i < block_num; i++) {
        page_talbe[i] = -1;
    }

    for(int i = 0; i < ADDRESS_COUNT; i++) {
        int hit = 0;
        int index = -1;
        for(int j = 0; j < block_num; j++) {
            //直接写入
            if(page_talbe[j] == -1) {
                index = j;
                break;
            }
            //命中
            else if(page_talbe[j] == page_sequence[i]) {
                hit = 1;
                break;
            }
        }
        //未命中
        if(hit == 0) {
            page_fault++;
            int max = 0;
            //index未赋值说明不可以直接写入，需要页面置换
            if(index == -1) {
                for(int j = 0; j < block_num; j++) {
                    int k = i + 1;
                    int flag = 0;
                    //查找后续指令所需当前内存块中的页面并找到最后的那一个
                    while(k < ADDRESS_COUNT) {
                        if(page_talbe[j] == page_sequence[k]) {
                            if(k > max) {
                                max = k;
```

```
                                index = j;
                            }
                            flag = 1;
                            break;
                        }
                        k++;
                    }
                    if(flag == 0) {
                        index = j;
                        break;
                    }
                }
            }

            page_talbe[index] = page_sequence[i];
        }
    }
    return page_fault;
}
```

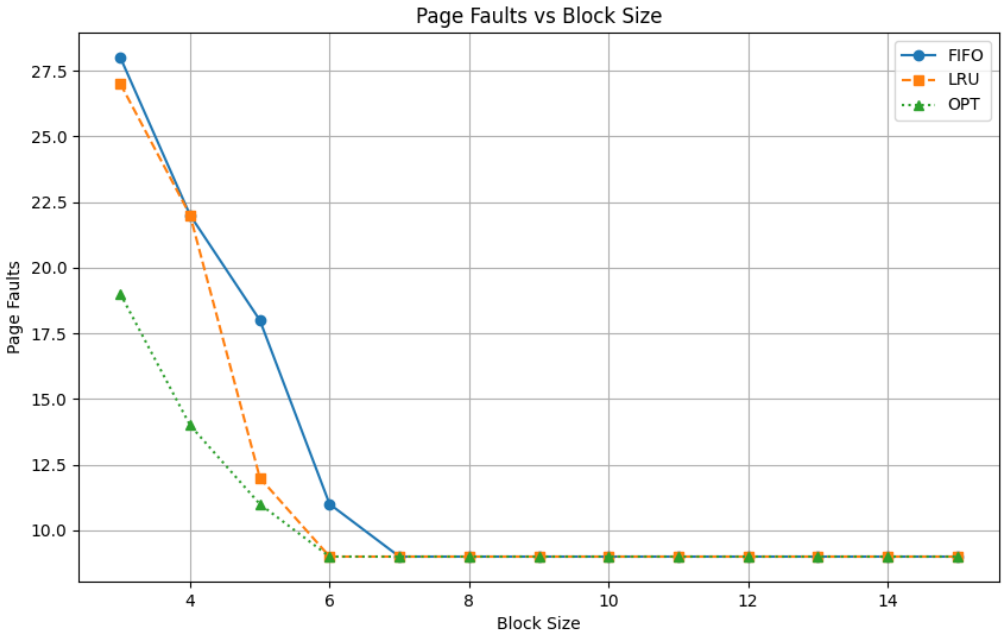
5 结果分析

对于100条指令，每个页面可以存10条指令，我们进行50次测试，可以得到如下图：



由于缺页率 = 页面失效次数/页地址流长度，而页地址流长度在本次实验中固定为100，因此缺页率直接以页面失效次数来代替，可以看到，由于题目所给数据限制，所以仅有10%的概率会用到前面所使用过的页面，所以`LRU算法`的替换策略是会比较低效的，大体上的缺页率是会和`FIFO算法`相同的，所以`FIFO调度算法`和`LRU调度算法`的缺页率是比较接近的，而且`Optimal算法`的缺页率是显著低于另外两个算法的。

接下来我们来分析在同样的内存访问串上执行，分配的物理内存块数量和缺页率之间的关系，生成的数据图如下：



可以看到对于同一个内存访问串，分配的物理内存块数量从3个到15个，随着分配的物理内存块数量增多，三种算法的缺页率都呈下降趋势，且`最优置换算法`的缺页率一直保持在最低，然后是`LRU算法`，再然后是`FIFO算法`。

6 实验总结

本次实验我实现了最优置换算法，先入先出算法以及最近最少使用算法，并将它们的性能加以比较，实验结果显示了最优置换算法的优异性能，然而在实际使用中，由于很难知道后续地址访问，最优置换算法只能用作参考，一般情况下，最近最少使用算法的缺页率会比先进先出算法低，而在特殊情况（如本次实验所要求的条件）下，先进先出算法和最近最少使用算法的缺页率会非常接近。

本次将课堂内容和理论实践相结合，使我对操作系统课程的内存管理这一块知识有了更深刻的理解，收获颇丰。