页面置换实验文档

陈思翰 19231258

前言

实验要求

本次页面置换竞争实验要求完成页面置换的模拟，并且会通过三项指标（时间，空间，缺页率）来判断页面置换操作的好坏，最终目的就是使三项指标都尽可能好（由于权值不同而有所侧重）。

回忆了一下，收获还是挺大的：

- 掌握了C++基本特性以及相比C多出的常用语法、常用STL的使用方法，了解了与java的不同
- 学会了CLION的基本使用，包括一些环境的配置，管道的配置，CMAKELIST的写法等等，最终用CLION实现了页面置换的评测机
- 复习了课堂上基本的页面置换算法，对于其优劣势有了更深的理解，同时上网查阅大量资料，学到了相应算法的改进算法
- 手动实现了一系列的容器，考虑了各种操作的时间、空间复杂度，将数据结构的知识进行了具体应用
- 掌握了优化相关技巧，包括register、inline等技巧

一、算法设计思想

$$Score = \frac{t - t_{max}}{t_{min} - t_{max}} * 0.2 + \frac{m - m_{max}}{m_{min} - m_{max}} * 0.1 + \frac{c - c_{max}}{c_{min} - c_{max}} * 0.7 \tag{1}$$

由rank标准可知，本次实验需要遵循以下原则：

- **页面命中率高于一切**，不能容忍优化其他指标所带来的缺页增加
- 对**时间成本优化的优先级**高于对**空间成本优化的优先级**

在实验开始之前，我坚信算法的好坏基本决定本次实验的成败，回顾课堂上讲过的具有可操作性的朴素页面置换算法，包括：`FIFO`，`LRU`，`CLOCK`，`LFU`等，发现所有的算法都有劣势，这里不再一一赘述，仔细权衡之后发现`LFU`、`LRU`是一个比较好的选择（可操作性，效果好坏）但是会有一个问题：

- `LRU`较短视，他不是根据长期的访问行为而建立访问模式的，而是根据短期访问行为而建立访问模式，对于之前经常访问并且往后还会经常访问的页面，LRU可能会因为短期没有访问到而将其错误地换出。比如存在周期性、偶发性访问的话，`LRU`将会造成大量缺页
- `LFU`需要历史记录，适应需要时间，即需要一定的时间根据之前的大量访问记录来决定留下哪些页面（访问频率最高），这样导致对于新加入的页面会因为缺乏建立时间，即便之后会被经常访问，但是由于不能马上建立模式，会被错误地置换出去

之后以LRU为突破口我上网搜索发现了几种优化算法：

`LRU-K` `2Q` `MQ` `LIRS` `ARC`等等，均对LRU做出了改进,经过一番尝试最终选择了`ARC`算法

因此主要介绍`ARC`算法：

从上面分析可知：LRU和LFU几乎是优势互补的，针对不同类型的数据分别有较好的表现，因此一个很自然的想法是：

能不能把`LRU`和`LFU`结合一下呢？（虽然在某些情况下可能会损失各自原有优势，但总体上更优）

`ARC`就是这样一个算法：

- 首先将给定的cache先平均分成两半，左边一部分对应LRU算法列表，右边对应LFU算法列表，列表中的每个元素都有对应的实际物理页号字段，这样就建立了映射
- 两个列表**长度可变**
- 为两个列表分别再准备一个**固定长度**的列表，称为`LRUGhost`,`LFUGhost`用来存储被置换出去的页面所对应的元素
- 首次访问的时候插入LRU列表头部
- LRU列表满了则将末尾元素插入`LRUGhost`头部，`LRUGhost`满了还进行插入，则末尾元素移出
    - 若击中`LRULive`则将此元素放入`LFULive`，若击中`LFULive`则将此元素提到队首，另外`LFULive`与`LRULive`一样，若满则末尾元素移出，插入`LFUGhost`头部，若满再移出末尾元素
- 若击中`Ghost`则移到对应`Live`头部,若满则同前一样进行调整

  整个过程中需要注意，`Ghost`长度是固定不变的，另外`Live`的长度是动态调整的，而这恰恰也是其自适应性的来源

综上，算法具有自适应性：

能根据当前访问页面的行为动态调整队列长度，以适应当前访问行为，减少缺页

二、算法实现技巧

1.建立相应的类

```
class Block {
public:
    long vpage;
    int ppage;
    Block *pre;
    Block *next;
    Block();
};
Block::Block() {
    vpage = 0;
    ppage = 0;
    pre = NULL;
    next = NULL;
}
//Live 中对应的元素结构
class GBlock {
public:
    long vpage;
    GBlock *pre;
    GBlock *next;
    GBlock();
};
GBlock::GBlock() {
    vpage = 0;
    pre = NULL;
    next = NULL;
}
//Ghost 中对应的元素结构
```

2.将具有高度相似性的操作内聚

ARC算法两侧几乎相同，可以提取共同的操作，再设不同的标志加以实现：

```
int changeCapacity(int choice) {
    p += choice;
    return 1;
}
//改变Live长度
int setEmptyPage(long *physic_memery,long vpage) {
    for (register int i = 0; i < MAX_PHY_PAGE; ++i) {
        if (EmptyPage[i] == 0) {
            physic_memery[i] = vpage;
            EmptyPage[i] = 1;
            return i;
        }
    }
    return -1;
}
void clearPhysic(int i) {
    EmptyPage[i] = 0;
}
//填入、清空cache
void removeGListTail(int choice,GBlock *gb)
//Ghost 末尾元素出队
void removeListIn(int choice,Block *b)
//击中了Live里面的元素 需要移出
void removeGListIn(int choice,GBlock *gb)
//击中了Ghost里的元素 需要移出
void getFront(Block *b)
//击中了Live里的元素 需要提到最前面
void insertListHead(int choice,Block *b)
//头插Live
void insertGListHead(int choice,GBlock *gb)
//头插Ghost
void freeListTail(int choice,long *physic_memery)
//移除Live末尾元素
void freeGListTail(int choice)
//移除Ghost末尾元素
//具体实现过于冗长不再展示
```

总的来说实现起来逻辑比较**复杂**,涉及到很多逻辑嵌套（当时自己也de了很久的bug）

另外就是链表的勾链操作等等，需要注意一下顺序和勾链的两端是否为空，注意考虑各种情况，不要写错了，具体操作不再赘述，详见源码

三、竞争实验过程中的优化与改进

1.数据结构

从上面算法的操作可知，涉及到大量的入队出队，因此选择链表进行实现，为了操作方便，构造双向链表，并为链表设置头部、尾部：

```
void init() {
    LRUHead->next = LRUTail;
```

```
    LRUTail->pre = LRUHead;
    LFUHead->next = LFUTail;
    LFUTail->pre = LFUHead;
    LRUGHead->next = LRUGTail;
    LRUGTail->pre = LRUGHead;
    LFUGHead->next = LFUGTail;
    LFUGTail->pre = LFUGHead;
}

//初始化链表
int LRUCnt;
int LFUCnt;
int LRUGCnt;
int LFUGCnt;
//由于ARC关注队列长度，在用链表实现的前提下，需要记录元素个数
```

还尝试过将64个物理页框也选择用链表管理，但是最后发现时间成本更大了，推测可能是页框数量比较小，进行链表维护操作的时间成本高于直接进行循环访问数组的时间成本，因此最后还是选择数组进行管理

```
int EmptyPage[64];
```

另外由于我在算法的实现过程中设置了两种类，不同的队列中存有不同的类`Live`中为`Block`,`Ghost`中为`GBlock`

两者之间的转换涉及到类中字段的转移，值得注意的是，只有`Live`和`Ghost`之间是需要做类的转化的，在实现的时候要注意一下

2.cpp优化技巧

网上了解到大量优化技巧，经过尝试发现比较有用的有一下几个：

```
const int MAX_PHY_PAGE (64);
//1 将define改为const 并且用这种括号初始化
inline int changeCapacity(int choice) {
    p += choice;
    return 1;
}
//2 简单函数使用内联
for (register int i = 0; i < MAX_PHY_PAGE; ++i) {
        if (EmptyPage[i] == 0) {
            physic_memery[i] = vpage;
            EmptyPage[i] = 1;
            return i;
        }
}
//3 把循环中常使用到的量 放入register 实现加速访问 效果显著
//4 变量的生明尽量靠近即将要使用的地方 即合适位置
```

四、本地测试情况

主要测试思路：

python自动生成随机数据，并通过命令行自动导入.exe文件进行测试

发现bug后，通过CLion管道重定向导入，并通过CLion~~舒服~~的工具debug

（由于数据量极大，想再不改变原始数据的情况下debug是难度极大的，好在CLion有友好的断点设置以及定位工具）

1.测试函数

```
long test[1700000];
int main() {
    //change !!!
    long pm[MAX_PHY_PAGE] = {0};
    int find = 0;
    long num;
    int loop;
    int i = 0;
    clock_t start,end,duration = 0;
    cin >> loop;
    while (i < loop) {
        cin >> num;
        test[i++] = num;
    }
    for (int i = 0; i < loop; ++i) {
        long page = test[i];
        long vpage = get_Page(page);
        start = clock();
        round = i;
        find = 0;
        pageReplace(pm,page);
        printf("%d\n",i);
        if (WA == 1) {
            printf("%d",i);
            return 1;
        }
        end = clock();
        duration += (end - start);
        for (int j = 0; j < MAX_PHY_PAGE; ++j) {
            //change !!!
            if (pm[j] == vpage) {
                find = 1;
                break;
            }
        }
        if (find == 0) {
            printf("%d WA \n",i + 1);
            cout<<"WAWAWAWAWAWAWAWAWAWAWAWA not in CACHE!!! END";
            return 0;
        }
//        if ()
//        printf("the %d is :\n",i + 1);
//        for (int j = 0; j < LRUGList.size(); ++j) {
//            printf("%d ",LRUGList.at(j));
//        }
//        printf("\n");
//        for (int j = 0; j < LRUList.size(); ++j) {
//            printf("%d ",LRUList.at(j).vpage);
//        }
//        printf("\n");
//        for (int j = 0; j < LFUList.size(); ++j) {
//            printf("%d ",LFUList.at(j).vpage);
```

```
//         }
//         printf("\n");
//         for (int j = 0; j < LFUGList.size(); ++j) {
//             printf("%d ",LFUGList.at(j));
//         }
//         printf("\n");
    }
    cout << "time cost: " << duration << endl;
    printf("%d : now the fault is %d\n",loop,fault);
    return 0;
}
```

正确性保证之后简化为时间成本测试函数：

```
long test[1700000];
int main() {
    long pm[MAX_PHY_PAGE] = {0};
    long num;
    int loop;
    int i = 0;
    clock_t start,end,duration = 0;
    cin >> loop;
    while (i < loop) {
        cin >> num;
        test[i++] = num;
    }
    for (int i = 0; i < loop; ++i) {
        long page = test[i];
        start = clock();
        pageReplace(pm,page);
        end = clock();
        duration += (end - start);
    }
    cout << "time cost: " << duration << endl;
    return 0;
}
```

2.数据生成与命令行大批量测试

```
import random
import subprocess
num = 1700000
MAXINT = 2147483647
loop = 1
print("是否要产生数据 Y N")
gen = input()
if gen == "Y":
    for i in range(loop):
        dataName = ".\\data" + str(i) + ".txt"
        with open(dataName, "w") as f:
            f.write(str(num) + "\n")
            for j in range(num):
                page = random.randint(4096,2147483647)
                f.write(str(page) + "\n")
    print("data gened\n")
outName = ".\\out.txt"
```

```
with open(outName,"w") as f:
    f.write("1start:\n")
for i in range(loop):
    dataName = "data" + str(i) + ".txt"
    cmd = "ARCchain.exe < " + dataName
    out = open(outName,"a")
    process = subprocess.Popen(cmd,shell=True,stdout=out)
    process.wait()
```

六、最终成果

	id	cost	instr	mem	score	rank
2	19231258	249456	229833975933.40	12.33328	0.991429	1
106	19373157	478545	232329992161.80	12.36453	0.207018	105
107	19373661	2305410	233616159439.80	12.32938	-5.21467	106
108						

七、对实验的意见

- 如果有页面号为0的情况请做好说明，如果觉得这个需要自己判断也请不要注明“不会出现页面号为0”的情况。在做的过程中，由于过于相信实验指导书导致出bug浪费了大量的时间……
- 页面置换实验完成的标准其实可以以三项指标加权的结果达到某个值为标准（比如说从明年开始取今年第30名的三项指标的加权值），这样既可以保证成功完成实验的同学所做出的成果的质量，又能避免意义不太大的内卷。（比如说我做的时自己必须时刻关注排名，开始是每天更新一次，后来会冷不丁地突然测一次，因为大家都在不断优化自己的代码，如果自己停止前进，说不定就掉出了前三十名），甚至还存在不少“高玩”大佬直到实验结束之前一秒都只交一个最简单的算法实现版本，等到最后再交上自己在本地优化了很久的最终版本……如果说这项竞争实验的最终目的就是选出前30名，那今年的做法很好（也只能这样做），如果最终目的是为了让大家多探索多学一点知识，锻炼一下自己的能力，我想做法可欠佳。