# 页面置换算法 (FIFO / LRU)

目录

[页面置换算法 (FIFO / LRU) 1](#_Toc502496888)

[一、实验环境 1](#_Toc502496889)

[二、相关知识 1](#_Toc502496890)

[1、页面置换 1](#_Toc502496891)

[2、页面置换算法 1](#_Toc502496892)

[三、实现源代码 2](#_Toc502496893)

[四、实验结果 4](#_Toc502496894)

**班级：2015211314**

**学号：2015211527**

**姓名：罗暄澍**

## 一、实验环境

本实验使用Win32环境，由C++语言实现。故采用如下环境

|  |  |
| --- | --- |
| 操作系统 | Windows 10 16299.125 |
| 编译环境 | GCC v6.3.0 |

## 二、相关知识

### 1、页面置换

增加多道程序的程度会导致内存的 过度分配（over-allocating) 。I/O 缓存也需要使用大量内存，缓存的使用会增加内存分配算法的压力，有的操作系统为 I/O 缓存分配了一定比例的内存，有的则允许用户进程和 I/O 子系统竞争全部系统内存。内存的过度分配会导致某个进程触发了页错误而没有空闲帧可用，因为按需调页对用户而言透明，因此操作系统不应当直接终止进程（操作系统也可以交换出一个进程来降低多道程序的级别，这种选择有时是好的）。

页置换（page replacement）发生在需要调页而没有空闲帧的情况，流程如下：

* 查找所需页在磁盘上的位置
* 查找空闲帧，若有则使用，否则通过页置换算法选择一个 牺牲（victim） 帧并将牺牲帧的内容写入备份存储（磁盘，交换空间），改变页表和帧表
* 将所需页写入到空闲帧，并改变页表和帧表
* 重启用户进程

可以看出，在没有帧空闲，需要页置换的情况下，会有两个页传输（一页换出，一页换入），这加倍了页错误处理时间。这一点可以通过在页表的每个条目中增加修改位（modify bit）或脏位（dirty bit）来降低额外开销，如果被置换出的页对应的修改位没有被设置，则说明此页从被调入内存后没有被修改，因此不必被写入回磁盘。

按需调页需要开发帧分配算法（frame-allocation algorithm）和页置换算法（page-replacement algorithm）。页置换算法的好坏可以计算页错误率评估：对于一个特定的内存地址引用序列，运行置换算法，计算出页错误的数量。这个引用序列称为引用串（reference string），可以人工产生也可以跟踪一个真实的系统并记录其访问内存地址。利用两个事实可以降低引用串的数据量：只考虑内存引用的页码而不考虑完整地址；如果有对页p的引用，则紧跟着对页p的引用绝不会产生页错误。

理论上来说，我们期待增加可用帧（增加物理内存大小就会增加可用帧数量）的数量能够使页错误的数量相对应减少。 Belady异常（Belady’s anomaly）指违背这一期待的现象：对于有的页置换算法，页错误率甚至可能随着分配的物理帧数增加而增加。

### 2、页面置换算法

#### (1) FIFO页置换

最简单的页置换算法，操作系统记录每个页被调入内存的时间，当必需置换掉某页时，选择最旧的页换出。实际操作中不需要真的记录调入时间，可以通过一个 FIFO 队列管理内存中的页，置换算法从队列的头部取出换出的页，将换入的页加入到队列尾部。FIFO 页置换算法的性能并不总是很好，它置换出的页可能是一个很久以前现在已经不再使用的页（符合我们的期望），也可能是一个进程创建时初始化的变量，而这个变量仍然在不停地被使用，此时被调出的这页很快就会再次导致页错误。

#### (2) 最优置换(optimal page-replacement)

是所有页置换算法中页错误率最低的，但它需要引用串的先验知识，因此无法被实现。它会将内存中的页 P 置换掉，页 P 满足：从现在开始到未来某刻再次需要页 P，这段时间最长。也就是 OPT 算法会 置换掉未来最久不被使用的页 。OPT 算法通常用于比较研究，衡量其他页置换算法的效果。

#### (3) 最近最少使用算法(least-recently-used algorithm)

简称 LRU，它置换掉到目前时刻最久未被使用的页。这一算法可以视作 OPT 的倒转，LRU 和 OPT 算法都属于 栈算法（stack algorithm） ，它们绝不会产生 Bleady 异常。一个有意思的地方在于，对于引用串 S，LRU 算法计算 S 和 S^R 的错误率时相同的（S^R 是引用串 S 的逆序），这一特性对 OPT 算法也满足。LRU 策略可能需要一定的硬件支持，因为它需要为页帧按上次使用时间确定一个排序序列。两种实现方式：

* 计数器（counters）

每个页表的条目关联一个时间域，CPU 增加一个计数器，每次内存引用发生时，计数器增加，并且将引用的页在页表中对应的条目的时间域更新为计数器的内容。这样 LRU 需要搜索页表置换具有最小时间域的页。这种方式每次内存访问都要写入内存，页表改变（因为 CPU 调度）的时候还需要保持时间，还需要考虑时钟溢出问题。

* 栈（stack）

采用页码栈维护，每当引用了一个页就将该页从栈中删除并放置到顶部，这样栈顶总是最近使用的页，栈底则为 LRU 需要替换的页。因为要从栈中删除某项，所以可实现为带有头指针和尾指针的双向链表，从栈中删除一页并放置到栈顶最坏情况下需要修改 6 个指针，但这种实现方式不需要搜索整个表。

### 三、实现源代码

#include<bits/stdc++.h>

using namespace std;

//如果能够命中，返回命中位置，不能则返回-1

int replace(vector<int> & frames,int data)

{

for(int i=0;i<frames.size();++i)

if(frames[i]==data)

return i;

return -1;

}

//输出frames

void printFrames(vector<int> & frames)

{

for(int frame:frames)

cout<<frame<<" ";

cout<<endl;

}

void pageReplaceFIFO(vector<int> nums,int size)

{//数据 页帧数

vector<int>frames(size,-1);//初始化为-1

int ptr=0,cnt=1;

for(int num:nums){

if(replace(frames,num)!=-1)

++cnt;

else

frames[ptr]=num,ptr=(ptr+1)%size;

printFrames(frames);

}

cout<<"Page fault: "<<nums.size() - cnt<<" times"<<endl;

}

void pageReplaceLRU(vector<int> nums,int size)

{//数据 页帧数

vector<int>frames(size,-1);//初始化为-1

vector<int>flag(size,0);

int ptr,cnt=1,time=0,isFull=0;

for(int num:nums){

if(time>=size)

isFull=1;

ptr=replace(frames,num);

if(ptr==-1){

if(isFull)

ptr=min\_element(flag.begin(),flag.end())-flag.begin();

else//如果是非满状态则逐个填入，恰好使用time作为计数器

ptr=time;

frames[ptr]=num;

}

else

++cnt;

flag[ptr]=time++;//无论是否命中都更新最近命中时间

printFrames(frames);

}

cout<<"Page fault: "<<nums.size() - cnt<<" times"<<endl;

}

void init(vector<int> & nums,int &size)

{

int len;

srand(time(NULL));

cout<<"请输入随机序列长度，页帧数量"<<endl;

cin>>len>>size;

nums.clear();

for(int i=0;i<len;++i)

nums.push\_back(rand()%10);

}

int main()

{

vector<int> nums{7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int size=3;

//vector<int> nums;

//int size;

//init(nums,size);

pageReplaceFIFO(nums,size);

pageReplaceLRU(nums,size);

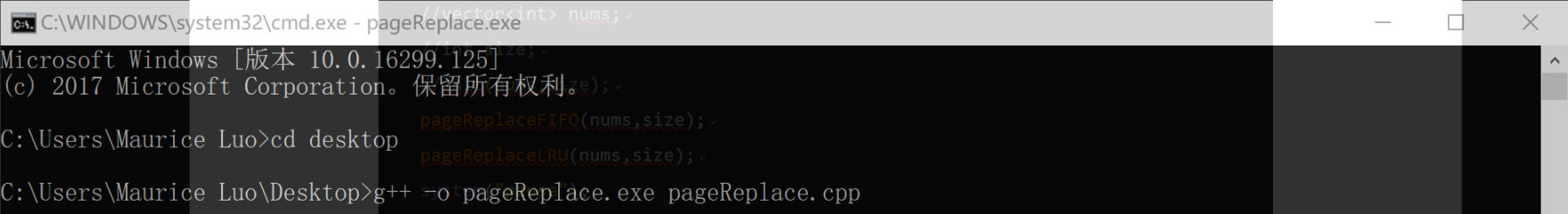
system("pause");

}

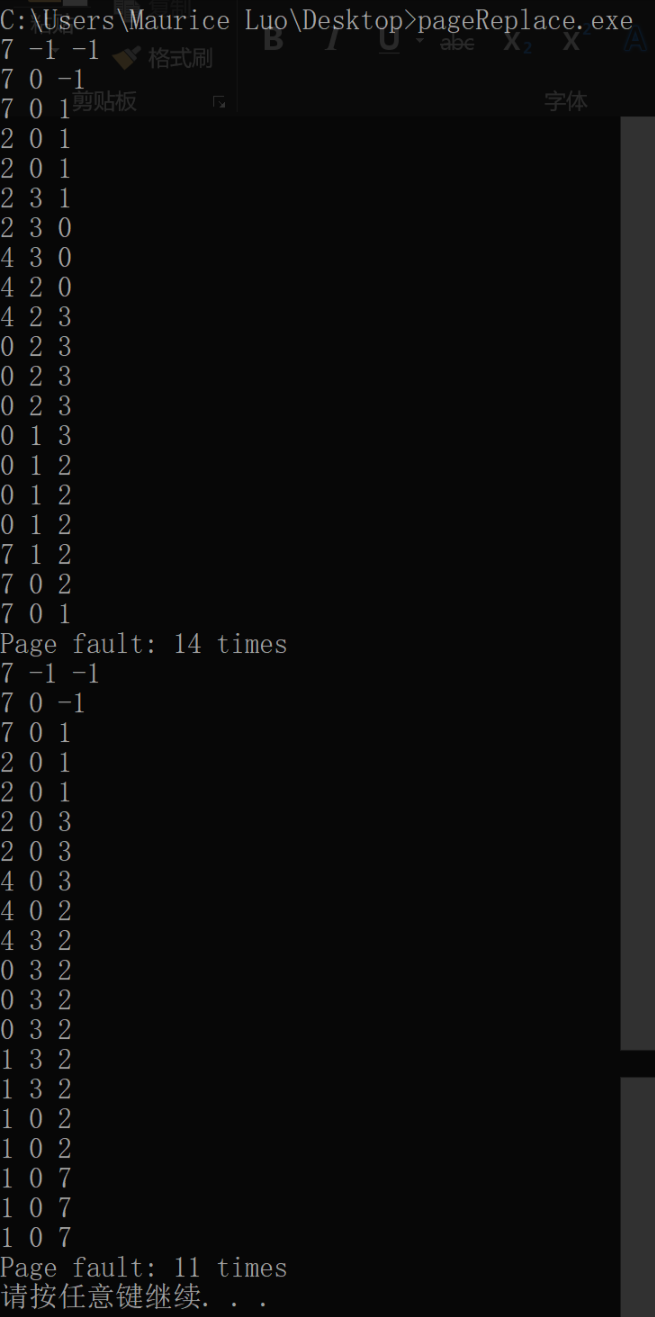
## 四、实验结果

若使用测试数据

### 1. 编译：



### 2. 运行结果(FIFO):



本程序还支持自定义数据集。源代码见附件。