**广州大学学生实验报告**

**开课学院及实验室：**计算机科学与网络工程学院软件实验室 **2023年 5月 日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | **计算机科学与网络工程学院** | **年级/专业/班** | **软件211** | **姓名** | 张景致 | **学号** | 32106300004 |
| **实验课程名称** | 操作系统实验 | | | | | **成绩** |  |
| **实验项目名称** | 进程管理与进程通信 | | | | | **指导老师** |  |

**实验三 内存管理**

**一、实验目的**

1、了解虚拟存储技术的特点，掌握虚拟存储请求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、了解程序设计技术和内存泄露的原因

1. **实验内容**

1、模拟实现请求页式存储管理的几种基本页面置换算法  
（1）最佳淘汰算法（OPT）

（2）先进先出的算法（FIFO）

（3）最近最久未使用算法（LRU））

1. **实验原理**

1、虚拟存储系统

UNIX中，为了提高内存利用率，提供了内外存进程对换机制；内存空间的分配和回收均以页为单位进行；一个进程只需将其一部分（段或页）调入内存便可运行；还支持请求调页的存储管理方式。

当进程在运行中需要访问某部分程序和数据时，发现其所在页面不在内存，就立即提出请求（向CPU发出缺中断），由系统将其所需页面调入内存。这种页面调入方式叫请求调页。

为实现请求调页，核心配置了四种数据结构：页表、页框号、访问位、修改位、有效位、保护位等。

2、页面置换算法

当CPU接收到缺页中断信号，中断处理程序先保存现场，分析中断原因，转入缺页中断处理程序。该程序通过查找页表，得到该页所在外存的物理块号。如果此时内存未满，能容纳新页，则启动磁盘I/O将所缺之页调入内存，然后修改页表。如果内存已满，则须按某种置换算法从内存中选出一页准备换出，是否重新写盘由页表的修改位决定，然后将缺页调入，修改页表。利用修改后的页表，去形成所要访问数据的物理地址，再去访问内存数据。整个页面的调入过程对用户是透明的。

（1）最佳淘汰算法（OPT）：选择永不使用或在未来最长时间内不再被访问的页面予以替换。

（2）先进先出的算法（FIFO）：选择在内存中驻留时间最久的页面予以替换。

（3）最近最久未使用算法（LRU）：选择过去最长时间未被访问的页面予以替换。

3、首先用srand( )和rand( )函数定义和产生指令序列，然后将指令序列变换成相应的页地址流，并针对不同的算法计算出相应的命中率。

（1）通过随机数产生一个指令序列，共320条指令。指令的地址按下述原则生成：

A：50%的指令是顺序执行的

B：25%的指令是均匀分布在前地址部分

C：25%的指令是均匀分布在后地址部分

具体的实施方法是：

A：在[0，319]的指令地址之间随机选取一起点m

B：顺序执行一条指令，即执行地址为m+1的指令

C：在前地址[0,m+1]中随机选取一条指令并执行，该指令的地址为m’

D：顺序执行一条指令，其地址为m’+1

E：在后地址[m’+2，319]中随机选取一条指令并执行

F：重复步骤A-E，直到320次指令

（2）将指令序列变换为页地址流

设：页面大小为1K；

用户内存容量4页到32页；

用户虚存容量为32K。

在用户虚存中，按每K存放10条指令排列虚存地址，即320条指令在虚存中的存放方式为：

第 0 条-第 9 条指令为第0页（对应虚存地址为[0，9]）

第10条-第19条指令为第1页（对应虚存地址为[10，19]）

………………………………

第310条-第319条指令为第31页（对应虚存地址为[310，319]）

按以上方式，用户指令可组成32页。

1. **实验中用到的系统调用函数**

因为是模拟程序，可以不使用系统调用函数。

1. **实验步骤**

1、 画出每个页面置换算法流程图；

2、 对算法所用的数据结构进行说明；

3、 测试数据随机产生。不可手工输入；

4、 编写程序并调试；

5、 多次测试程序，截屏输出实验结果；

6、 根据实验结果与理论课讲述的原理进行实验分析。

**六、思考题**

1、从几种算法的命中率看，哪个算法最高？哪个算法最低？对每个页面的执行结果进行分析。

2、OPT算法在执行过程中可能会发生错误，为什么？

**七、实验总结**

**八、源代码**

#include <stdio.h>  
#include <stdbool.h>  
#include <stdlib.h>  
  
#include "lcpr.h"  
  
#define **loop** while(**true**)  
  
*// --- Default Configuration ---  
//const int ins\_count = 320;  
//const int ins\_per\_page = 10;  
//const int cache\_size\_min = 4;  
//const int cache\_size\_max = 32;*const int ins\_count = 320;  
const int ins\_per\_page = 10;  
const int cache\_size\_min = 4;  
const int cache\_size\_max = 32;  
  
void populate\_prophecy(int \*seq) {  
#define **tick\_tock** index += 1; if (index == ins\_count) break;  
 int index = 0;  
 **loop** {  
 seq[index] = lcpr\_rand\_int\_bounded(ins\_count);  
 **tick\_tock** seq[index] = seq[index - 1] + 1;  
 **tick\_tock** seq[index] = lcpr\_rand\_int\_bounded(seq[index - 1]);  
 **tick\_tock** seq[index] = seq[index - 1] + 1;  
 **tick\_tock** seq[index] = seq[index - 1] + lcpr\_rand\_int\_bounded(ins\_count - seq[index - 1]);  
 **tick\_tock** }  
}  
  
#define **page\_of**(addr) (addr / ins\_per\_page)  
#define **evaluator**(name) int name (const int \*seq, int cache\_size)  
typedef int (\*evaluator\_t)(const int \*, int);  
  
**evaluator**(fifo) {  
 int cache[cache\_size];  
 int head = 0;  
 int miss\_count = 0;  
 for (int i = 0; i < ins\_count; ++i) {  
 int page = **page\_of**(seq[i]);  
 **bool** hit = **false**;  
 for (int j = 0; j < cache\_size; ++j) {  
 if (cache[j] == page) {  
 hit = **true**;  
 break;  
 }  
 }  
 if (!hit) {  
 cache[head] = page;  
 head = (head + 1) % cache\_size;  
 miss\_count += 1;  
 }  
 }  
 return ins\_count - miss\_count;  
}  
  
**evaluator**(lru) {  
 int cache[cache\_size];  
 int usage[cache\_size];  
 int miss\_count = 0;  
 for (int i = 0; i < cache\_size; ++i) {  
 cache[i] = -1;  
 usage[i] = 0;  
 }  
 for (int i = 0; i < ins\_count; ++i) {  
 int page = **page\_of**(seq[i]);  
 **bool** hit = **false**;  
 int lru\_index = 0;  
 for (int j = 0; j < cache\_size; ++j) {  
 if (cache[j] == page) {  
 hit = **true**;  
 usage[j] = i; *// update usage time for hit* break;  
 } else if (cache[j] == -1) { *// found empty slot* cache[j] = page;  
 usage[j] = i;  
 miss\_count += 1;  
 hit = **true**;  
 break;  
 } else if (usage[j] < usage[lru\_index]) { *// find least recently used index* lru\_index = j;  
 }  
 }  
 if (!hit) { *// replace least recently used page* cache[lru\_index] = page;  
 usage[lru\_index] = i;  
 miss\_count += 1;  
 }  
 }  
 return ins\_count - miss\_count;  
}  
  
**evaluator**(opt) {  
 int cache[cache\_size];  
 int miss\_count = 0;  
 for (int i = 0; i < ins\_count; ++i) {  
 int page = **page\_of**(seq[i]);  
 **bool** hit = **false**;  
 for (int j = 0; j < cache\_size; ++j) {  
 if (cache[j] == page) {  
 hit = **true**;  
 break;  
 }  
 }  
 if (!hit) {  
 int replace\_index = -1;  
 int farthest\_distance = -1;  
 for (int j = 0; j < cache\_size; ++j) {  
 int distance = 0;  
 for (int k = i + 1; k < ins\_count; ++k) {  
 if (cache[j] == **page\_of**(seq[k])) {  
 break;  
 }  
 ++distance;  
 }  
 if (distance > farthest\_distance) {  
 replace\_index = j;  
 farthest\_distance = distance;  
 }  
 }  
 cache[replace\_index] = page;  
 miss\_count += 1;  
 }  
 }  
 return ins\_count - miss\_count;  
}  
  
**evaluator**(noop) {  
 return 0; *// No-op*}  
  
void test(const int \*seq, const char \*name, evaluator\_t eval, int cache\_size) {  
 int hit\_count = eval(seq, cache\_size);  
 printf("R %s,%d,%d,%d\n", name, cache\_size, hit\_count, ins\_count);  
}  
  
int main() {  
 int seq[ins\_count];  
 lcpr\_init\_time();  
 populate\_prophecy(seq);  
 printf("--- PROPHECY ---\n");  
 for (int i = 0; i < ins\_count; ++i) {  
 printf("%d ", seq[i]);  
 }  
 printf("\n\n");  
  
 for (int size = cache\_size\_min; size <= cache\_size\_max; ++size) {  
 test(seq, "noop", noop, size);  
 test(seq, "opt", opt, size);  
 test(seq, "fifo", fifo, size);  
 test(seq, "lru", lru, size);  
 printf("\n");  
 }  
  
}