**某某大学 操作系统实验 实验报告**

实验名称： 存储管理

班 号：

姓 名：

学 号：

**一 实验目的**

通过模拟实现内存分配的伙伴算法和请求页式存储管理的几种基本页面置换算法，了解存储技术的特点。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

**二 实验内容**

**1.实现一个内存管理的伙伴算法，实现内存块申请时的分配和释放后的回收。**

实验准备

用随机函数仿真进程进行内存申请，并且以较为随机的次序进行释放。对其碎片进行统计，当申请分配内存失败时区分实际空间不足和由于碎片而不能满足。

2**.设计一个虚拟存储区和内存工作区，并使用下述算法计算访问命中率。**

1) 最佳置换算法（Optimal）

2) 先进先出法（Fisrt In First Out）

3) 最近最久未使用（Least Recently Used）

4) 最不经常使用法（Least Frequently Used）

5) 最近未使用法（No Used Recently）

其中，命中率＝１－页面失效次数／页地址流长度。

试对上述算法的性能加以较各：页面个数和命中率间的关系；同样情况下的命中率比较。

实验准备

本实验中主要的流程：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。

实验可先从一个具体的例子出发。

（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。

**三 项目分析**

**3.1 对于Buddy System**

Buddy System是一种经典的内存管理算法。在Unix和Linux操作系统中都有用到。其作用是减少存储空间中的空洞、减少碎片、增加利用率。避免外碎片的方法有两种：   
 a.利用分页单元把一组非连续的空闲页框映射到非连续的线性地址区间。   
 b.开发适当的技术来记录现存的空闲连续页框块的情况，以尽量避免为满足对小块的请 求而把大块的空闲块进行分割。   
基于下面三种原因，内核选择第二种避免方法：   
 a.在某些情况下，连续的页框确实必要。   
 b.即使连续页框的分配不是很必要，它在保持内核页表不变方面所起的作用也是不容忽视的。假如修改页表，则导致平均访存次数增加，从而频繁刷新TLB。   
 c.通过4M的页可以访问大块连续的物理内存，相对于4K页的使用，TLB未命中率降低，加快平均访存速度。   
 Buddy算法将所有空闲页框分组为10个块链表，每个块链表分别包含1,2,4,8,16,32,64,128,256,512个连续的页框，每个块的第一个页框的物理地址是该块大小的整数倍。如，大小为16个页框的块，其起始地址是16\*2^12的倍数。   
 例，假设要请求一个128个页框的块，算法先检查128个页框的链表是否有空闲块，如果没有则查256个页框的链表，有则将256个页框的块分裂两份，一 份使用，一份插入128个页框的链表。如果还没有，就查512个页框的链表，有的话就分裂为128，128，256，一个128使用，剩余两个插入对应链 表。如果在512还没查到，则返回出错信号。   
回收过程相反，内核试图把大小为b的空闲伙伴合并为一个大小为2b的单独块，满足以下条件的两个块称为伙伴：   
 a.两个块具有相同的大小，记做b。   
 b.它们的物理地址是连续的。   
 c.第一个块的第一个页框的物理地址是2\*b\*2^12的倍数。   
该算法迭代，如果成功合并所释放的块，会试图合并2b的块来形成更大的块。

为了模拟Buddy System算法，我采用了数的数据结构，并使用了结构体，来记录各项数据与标记，虽然不是真正的操作系统使用的方法，但成功模拟了插入和回收的过程。在回收时我采用物理上的合并——即删除实际的物理节点，释放空间。然而实际中可能根据需要仅仅是删除了标记项，使之标记成没用过的，从而避免了合并，会提高下一次的插入操作。

**碎片百分比 = 碎片总大小/总内存大小**

**3.2 页式存储管理中页面置换算法模拟设计**

页式虚拟存储器实现的一个难点是设计页面调度（置换）算法，即将新页面调入内存时，如果内存中所有的物理页都已经分配出去，就要按某种策略来废弃某个页面，将其所占据的物理页释放出来，供新页面使用。

页面替换算法主要用于如下几个地方：

　　(1) 虚拟存储器中，主存页面（或程序段）的替换。

　　(2) Cache中的块替换。

　　(3) 虚拟存储器的快慢表中，快表的替换。

　　(4) 虚拟存储器中，用户基地址寄存器的替换。

在虚拟存储器中常用的页面替换算法有如下几种：

(1) 最优替换算法，即OPT算法（OPTimal replacement algorithm）。上面介绍的几种页面替换算法主要是以主存储器中页面调度情况的历史信息为依据的，它假设将来主存储器中的页面调度情况与过去一段时间内主存储器中的页面调度情况是相同的。显然，这种假设不总是正确的。最好的算法应该是选择将来最久不被访问的页面作为被替换的页面，这种替换算法的命中率一定是最高的，它就是最优替换算法。

要实现OPT算法，唯一的办法是让程序先执行一遍，记录下实际的页地址流情况。根据这个页地址流才能找出当前要被替换的页面。显然，这样做是不现实的。因此，OPT算法只是一种理想化的算法，然而，它也是一种很有用的算法。实际上，经常把这种算法用来作为评价其它页面替换算法好坏的标准。在其它条件相同的情况下，哪一种页面替换算法的命中率与OPT算法最接近，那么，它就是一种比较好的页面替换算法。

(2) 先进先出算法，即FIFO算法（First-In First-Out algorithm）。这种算法选择最先调入主存储器的页面作为被替换的页面。它的优点是比较容易实现，能够利用主存储器中页面调度情况的历史信息，但是，没有反映程序的局部性。因为最先调入主存的页面，很可能也是经常要使用的页面。

(3) 最久没有使用算法，即LRU算法（Least Recently Used algorithm）。这种算法把近期最久没有被访问过的页面作为被替换的页面。它把LFU算法中要记录数量上的"多"与"少"简化成判断"有"与"无"，因此，实现起来比较容易。

　 (4) 近期最少使用算法，即LFU算法（Least Frequently Used algorithm）。这种算法选择近期最少访问的页面作为被替换的页面。显然，这是一种非常合理的算法，因为到目前为止最少使用的页面，很可能也是将来最少访问的页面。该算法既充分利用了主存中页面调度情况的历史信息，又正确反映了程序的局部性。但是，这种算法实现起来非常困难，它要为每个页面设置一个很长的计数器，并且要选择一个固定的时钟为每个计数器定时计数。在选择被替换页面时，要从所有计数器中找出一个计数值最大的计数器。因此，通常采用如下一种相对比较简单的方法。

(5) 最近未使用法（No Used Recently）

**缺页中断率 = 缺页中断次数/总的页面引用次数\*100%**

**四 问题实现及运行结果**

**4.1对于Buddy System**

**4.1.1 数据结构**

**节点的结构**

struct memory\_node

{

int remain\_max; //还能分配的最大空间

int use; //已用掉的大小

int total;

int pid; //标记进程号

int flag; //是否分裂，即是否有孩子,0为未分配

struct memory\_node \*left;

struct memory\_node \*right;

};

typedef struct memory\_node \*NODE;

**4.1.2 全局变量**

|  |
| --- |
| # define Inital 1024 //初始时的总内存 |
| NODE root=(memory\_node \*)malloc(1\*sizeof(memory\_node));//根节点 |
| int chip=0; // 记录总的碎片大小 |
| int chip\_num=0; //记录碎片的数量 |

**4.1.3 函数定义**

|  |  |
| --- | --- |
| 函数名称 | 实现功能 |
| void inital(NODE node,int num) | **初始化节点** |
| int memory\_alloc(int item,NODE node,int pid) | **伙伴算法的分配** |
| void Lprintf(void) | **输出格式控制** |
| void preoder\_print(NODE node) | **中序遍历打印出所有的叶节点** |
| int memory\_merge(int pid,NODE node) | **伙伴算法的合并** |

**4.1.4 依次插入item = 129 pid =0 item = 267 pid =1的示意图**

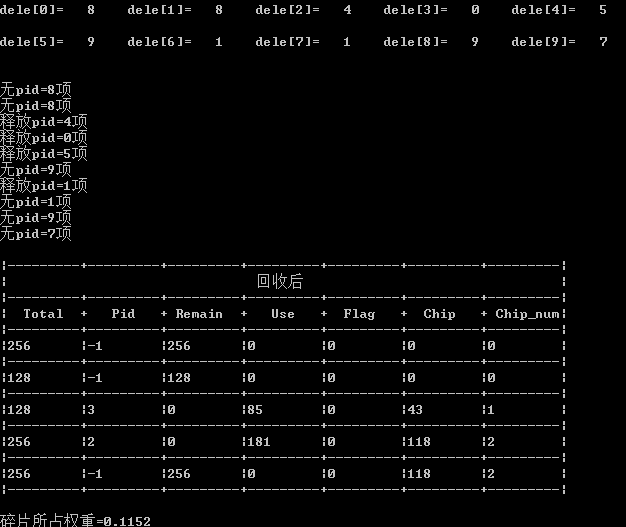
**4.1.5 删除 pid = 0的过程图**

**4.1.6 在window下运行的结果**

插入10个随机的值并分配结果如下，其中表中没有pid的为插入不成功的



释放随机的10个pid，找不到的不释放



**4.1.7 在linux下**

生成两个线程，使用mutex为锁，当一个在插入时，另一个不能释放，达到互斥的效果

**新增如下变量：**

pthread\_mutex\_t mutex;//信号量mutex

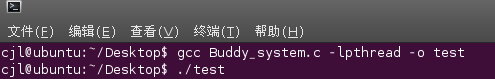
int pro\_speed=4,con\_speed=2;//可以设定两者速度

**新增函数：**

void \*producer(void \*param) //插入的线程函数

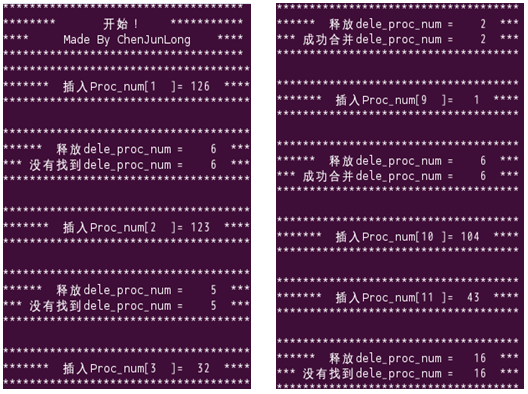
void \*consumer(void \*param)// 释放的线程函数

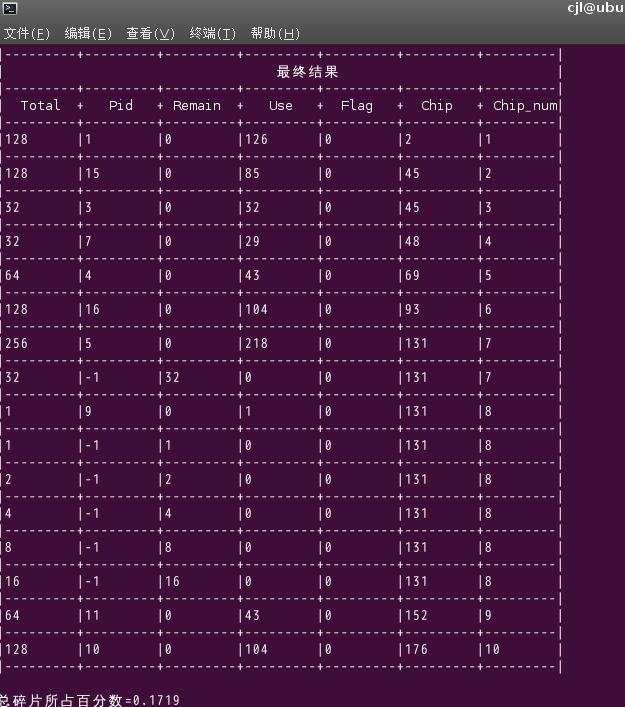
**首先： 编译链接运行**





**显示如下：**





**4.1.7 源代码见附录（linux下的）**

**4.2 页式存储管理中页面置换算法模拟设计**

**4.2.1 数据结构**

**(1)页面类型**

typedef struct

{

int pn,;//页号

int pn;// 面号

int counter;// 一个周期内访问该页面的次数

int time;// time为访问时间

}pl-type;

**(2) 页面控制结构**

Pfc\_struct

{

int pn,pfn;

struct pfc\_struct \*next;

}

typedef struct pfc\_struct pfc\_type;

|  |
| --- |
| pfc\_type pfc\_struct[total\_vp];// pfc[total\_vp]定义用户进程虚页控制结构 |
| pfc\_type \*freepf\_head;//空页面头的指针 |
| pfc\_type \*busypf\_head;//忙页面头的指针, |
| pfc\_type \*busypf\_tail;//忙页面尾的指针. |

**4.2.2 变量定义**

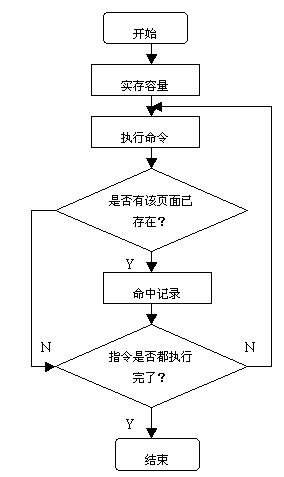
|  |  |
| --- | --- |
| int a[total\_instruction] | 指令流数据组 |
| int page[total\_instruction] | **每条指令所属的页号** |
| int offset[total\_instruction] | **每页装入10条指令后取模运算页号偏移值** |
| int total\_pf | **用户进程的内存页面数** |
| int disaffect | **页面失效次数** |

**4.2.3 函数定义**

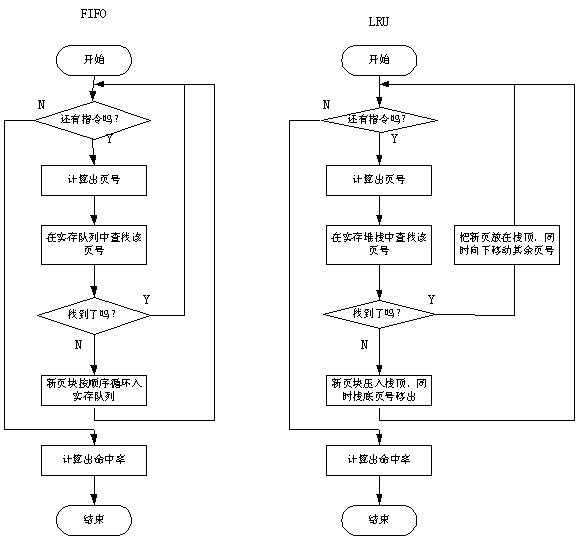
|  |  |
| --- | --- |
| void Lprintf(void) | 格式控制 |
| void initialize(int total\_pf) | **初始化相关数据结构** |
| void FIFO(int total\_pf) | **先进先出法（Fisrt In First Out）** |
| void LRU (int total\_pf) | **最近最久未使用（Least Recently Used）** |
| void OPT(int total\_pf) | **最佳置换算法（Optimal）** |
| void LFU(int total\_pf) | **最不经常使用法（Least Frequently Used）** |
| void NUR(int total\_pf ) | **最近未使用法（No Used Recently）** |

**4.2.4 流程图**

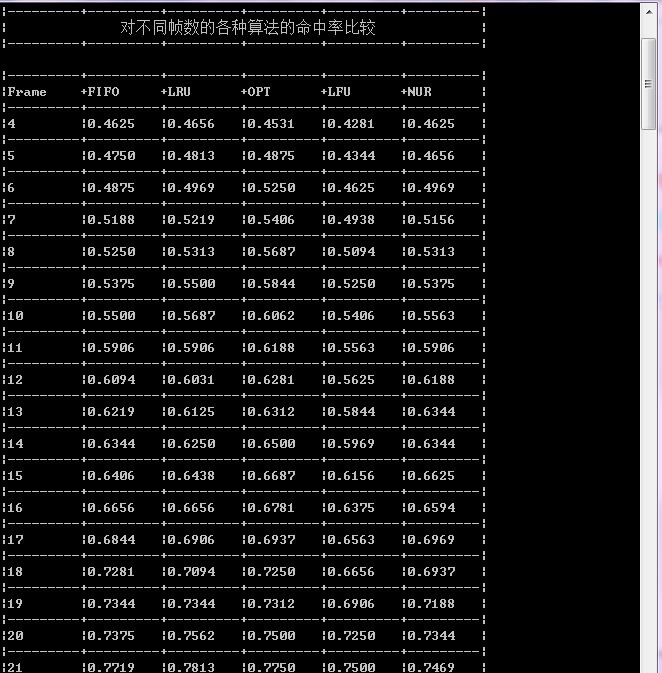
**(1) 页面调度模拟算法流程示例图**

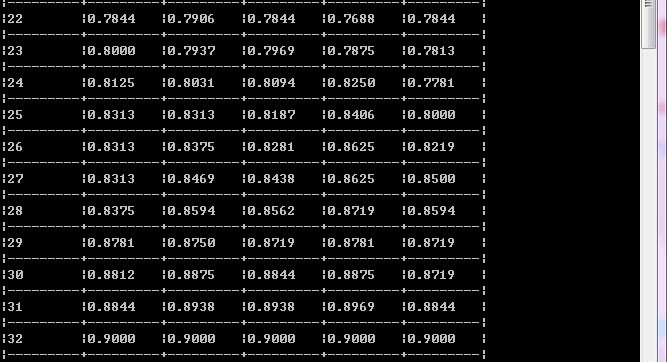


**(2) FIFO和LRU调度模拟算法示例图**



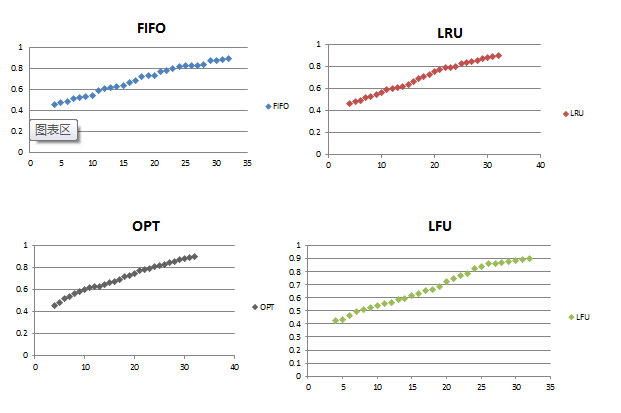
**4.2.5 运行实例**



****

**4.2.6 数据分析**

对各个算法进行统计分析得到如下表，前五个是单独的，最后一个合到一个图



由图表可知：

以OPT算法的命中率最高，NUR算法次之，其次是LFU算法和LRU算法，最后是FIFO算法。

**五 所遇到问题及实验感悟**

实验中对于伙伴算法编写释放函数时，一开始总是出现错误。后经过检查，发现原来是释放节点的判断函数出现了错误，进而使得最后总是全部释放完。在后来将其移植到linux下是也遇到了一些小问题，如编译不过、打印错误等等，不过最终都解决了。

通过对上述两个实验的实际操作，加深了对伙伴算法、也米娜

**六 附录：源代码**

**6.1 Buddy 算法**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <time.h>

# define Inital 1024

struct memory\_node

{

int remain\_max;

int use;

int total;

int pid;

int flag;

struct memory\_node \*left;

struct memory\_node \*right;

};

struct memory\_node p={0,0,0,0,0,NULL,NULL};

typedef struct memory\_node \*NODE;

NODE root=NULL;

int chip=0;

int chip\_num=0;

int proc\_num=0;

pthread\_mutex\_t mutex;//信号量mutex

int pro\_speed=4,con\_speed=2;//可以设定两者速度

**/\***

**Name: void inital(NODE node,int num)**

**Achieve: 初始化节点**

**\*/**

void inital(NODE node,int num)

{

node->remain\_max = num;

node->total = num;

node->use = 0;

node->flag = 0;

node->pid = -1;

node->left = NULL;

node->right = NULL;

}

**/\***

**Name: int memory\_alloc(int item,NODE node)**

**Achieve: 伙伴算法的分配**

**\*/**

int memory\_alloc(int item,NODE node,int pid)

{

int num,i,temp1,temp2;

if(item > node->remain\_max)

{

printf("\*\*\*\*\*无足够内存,请等待(执行下一条)\*\*\*\n");

return node->remain\_max;

}

if(node->flag==0)//如果还没有分裂

{

if(item > node->remain\_max/2)//如果item大于当前容量的1/2

{

node->remain\_max = 0;

node->use = item;

node->pid = pid;

return node->remain\_max ;

}

else //如果item小于当前容量的1/2,则应分裂节点

{

NODE l=(memory\_node \*)malloc(1\*sizeof(memory\_node));

NODE r=(memory\_node \*)malloc(1\*sizeof(memory\_node));

num = node->remain\_max/2;

inital(l,num);//初始化左孩子

inital(r,num);//初始化右孩子

node->left = l;

node->right = r;

node->remain\_max = num;//所能使用的最大值为node->remain\_max/2

node->flag = 1;//标记为1，表明是非叶节点

i = memory\_alloc(item,l,pid);//分配给左孩子

return num;

}

}

else//如果已分裂

{

temp1 = node->left->remain\_max;

temp2 = node->right->remain\_max;

if(temp1<=temp2)//若左边的remain小于右边的remain

{

if(item<=temp1)//如果item小于左边的

{

i = memory\_alloc(item,node->left,pid);//分配给左孩子

node->remain\_max = temp2;//当前的remain=temp2

return temp2;

}

else//如果item大于左边的

{

i = memory\_alloc(item,node->right,pid);//分配给右孩子

if(i > temp1)//当前的remain等于左右孩子remain的最大值

{

node->remain\_max = i;

return i;

}

else

{

node->remain\_max = temp1;

return temp1;

}

}

}

else//若左边的remain大于右边的remain

{

if(item<=temp2)//如果item小于右边的

{

i = memory\_alloc(item,node->right,pid);//分配给右孩子

node->remain\_max = temp1;

return temp1;

}

else

{

i = memory\_alloc(item,node->left,pid);//分配给左孩子

if(i >=temp2)

{

node->remain\_max = i;

return i;

}

else

{

node->remain\_max = temp2;

return temp2;

}

}

}

}

}

**/\***

**Name: void Lprintf(void)**

**Achieve: 输出格式控制**

**\*/**

void Lprintf(void)

{

int i,j;

printf("|");

for(i = 1;i<=7;i++)

{

for(j = 1;j<=9;j++)

printf("-");

if(i!=7)

printf("+");

}

printf("|\n");

}

**/\***

**Name: void preoder\_print(NODE node)**

**Achieve: 中序遍历打印出所有的叶节点**

**\*/**

void preoder\_print(NODE node)

{

if(node->flag==0)

{

if(node->total!=node->remain\_max)

{

chip\_num++;

chip+=node->total-node->use;

}

printf("|%-9d",node->total);

printf("|%-9d",node->pid);

printf("|%-9d",node->remain\_max);

printf("|%-9d",node->use);

printf("|%-9d",node->flag);

printf("|%-9d",chip);

printf("|%-9d",chip\_num);

printf("|\n");

Lprintf();

}

else

{

preoder\_print(node->left);

preoder\_print(node->right);

}

}

**/\***

**Name: void memory\_merge()**

**Achieve: 伙伴算法的合并**

**\*/**

int memory\_merge(int pid,NODE node)

{

int num = node->total;

int temp1,temp2;

if(node->left==NULL&&node->right==NULL)

{

if(node->pid==pid)

{

inital(node,num);

return -2;

}

else

return -1;

}

temp1 = memory\_merge(pid,node->left);

if(temp1==-2)

{

if(node->right->remain\_max!=num/2)

node->remain\_max = node->left->total;

else if(node->left->flag==0)

{

free(node->left);

free(node->right);

inital(node,num);

}

return -2;

}

else

{

temp2 = memory\_merge(pid,node->right);

if(temp2==-1)

return -1;

else

{

if(node->left->remain\_max!=num/2)

node->remain\_max = node->right->total;

else if(node->right->flag==0)

{

free(node->left);

free(node->right);

inital(node,num);

}

return -2;

}

}

}

**/\***

**Name: void \*producer(void \*param)**

**Achieve: 插入的线程函数**

**\*/**

void \*producer(void \*param)

{

int num,temp1;

int item;

while(1)

{

sleep(rand()%(16-pro\_speed)+1);

pthread\_mutex\_lock(&mutex);

item = rand()%300+1;

proc\_num++;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\*\*\*\*\*\* 插入Proc\_num[%-3d]=%4d \*\*\*\*\n",proc\_num,item);

temp1 = memory\_alloc(item,root,proc\_num);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\n");

root->remain\_max = temp1;

pthread\_mutex\_unlock(&mutex);

}

}

**/\***

**Name: void \*consumer(void \*param)**

**Achieve: 释放的线程函数**

**\*/**

void \*consumer(void \*param)

{

int num,temp1;

int dele\_proc\_num;

while(1){

sleep(rand()%(16-con\_speed)+1);

pthread\_mutex\_lock(&mutex);

dele\_proc\_num = rand()%(proc\_num+5)+1;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\*\*\*\*\* 释放dele\_proc\_num = %4d \*\*\*\n",dele\_proc\_num);

temp1 = memory\_merge(dele\_proc\_num,root);

if(temp1==-1)

printf("\*\*\* 没有找到dele\_proc\_num = %4d \*\*\*\n",dele\_proc\_num);

else

printf("\*\*\* 成功合并dele\_proc\_num = %4d \*\*\*\n",dele\_proc\_num);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

pthread\_mutex\_unlock(&mutex);

}

}

int main()//主函数

{

pthread\_t tid1,tid2;

pthread\_attr\_t attr1,attr2;

root=(NODE)malloc(1\*sizeof(p));

inital(root,Inital);

srand(time(NULL));

pthread\_mutex\_init(&mutex,NULL);//初始化

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\*\*\*\*\*\*\* 开始！ \*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\*\*\* Made By ChenJunLong \*\*\*\*\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

pthread\_attr\_init(&attr1);

pthread\_create(&tid1,&attr1,producer,NULL);

pthread\_attr\_init(&attr2);

pthread\_create(&tid2,&attr2,consumer,NULL);

sleep(100);

printf("\n");

Lprintf();

printf("| 最终结果 |\n");

Lprintf();

printf("| Total + Pid + Remain + Use + Flag + Chip + Chip\_num|\n");

Lprintf();

preoder\_print(root);

printf("\n总碎片所占百分数=%6.4f\n\n",(float)chip/Inital);

printf("\*\*\*\*\*\*\*程序over\*\*\*\*\*\*\*\n");

return 0;

}

**6.2** **页式存储管理中页面置换算法**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#define TRUE 1

#define FALSE 0

#define INVALID -1

#define NUL 0

#define total\_instruction 320 //指令流长

#define total\_vp 32 //虚页长

#define clear\_period 50 //清零周期

typedef struct

{

int pn; //页号

int pfn; // 面号

int counter; // 一个周期内访问该页面的次数

int time; // time为访问时间

}pl\_type;

pl\_type pl[total\_vp]; //页面结构数组

struct pfc\_struct{ //页面控制结构

int pn,pfn;

struct pfc\_struct \*next;

};

typedef struct pfc\_struct pfc\_type;

pfc\_type pfc[total\_vp],\*freepf\_head,\*busypf\_head,\*busypf\_tail;

int diseffect,a[total\_instruction];

int page[total\_instruction], offset[total\_instruction];

**/\***

**Name: void Lprintf(void)**

**Achieve: 格式控制**

**\*/**

void Lprintf(void)

{

int i,j;

printf("|");

for(i = 1;i<=6;i++)

{

for(j = 1;j<=9;j++)

printf("-");

if(i!=6)

printf("+");

}

printf("|\n");

}

**/\***

**Name: void initialize(int total\_pf)**

**Achieve:初始化相关数据结构**

**\*/**

void initialize(int total\_pf)

{

int i;

diseffect=0;

for(i=0;i<total\_vp;i++)

{

pl[i].pn=i;pl[i].pfn=INVALID; //置页面控制结构中的页号，页面为空

pl[i].counter=0;pl[i].time=-1; //页面控制结构中的访问次数为0，时间为-1

}

for(i=1;i<total\_pf;i++)

{

pfc[i-1 ].next=&pfc[i];pfc[i-1].pfn=i-1;//建立pfc[i-1]和pfc[i]之间的连接

}

pfc[total\_pf-1].next=NUL;pfc[total\_pf-1].pfn=total\_pf-1;

freepf\_head=&pfc[0]; //页面队列的头指针为pfc[0]

}

**/\***

**Name:void FIFO(int total\_pf)**

**Achieve:先进先出法（Fisrt In First Out）**

**\*/**

void FIFO(int total\_pf)

{

int i,j;

pfc\_type \*p;//中间变量

initialize(total\_pf); //初始化相关页面控制用数据结构

busypf\_head=busypf\_tail=NULL; //忙页面队列头，队列尾链接

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) //页面失效

{

diseffect+=1; //失效次数

if(freepf\_head==NULL)//无空闲页面

{

p=busypf\_head->next;

pl[busypf\_head->pn].pfn=INVALID;

freepf\_head=busypf\_head; //释放忙页面队列的第一个页面

freepf\_head->next=NULL; //表明还是缺页\*/

busypf\_head=p;

}

p=freepf\_head->next;

freepf\_head->pn=page[i];

pl[page[i]].pfn=freepf\_head->pfn;

freepf\_head->next=NULL; //使busy的尾为null

if(busypf\_tail==NULL)

{

busypf\_tail=busypf\_head=freepf\_head;

}

else

{

busypf\_tail->next=freepf\_head;

busypf\_tail=freepf\_head;

}

freepf\_head=p;

}

}

printf("%6.4f",1-(float)diseffect/320);

}

**/\***

**Name: void LRU (int total\_pf)**

**Achieve: 最近最久未使用（Least Recently Used）**

**\*/**

void LRU (int total\_pf)

{

int min,minj,i,j,present\_time; //minj为最小值下标

initialize(total\_pf);

present\_time=0;

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) //页面失效

{

diseffect++;

if(freepf\_head==NULL) //无空闲页面

{

min=32767;//设置最大值

for(j=0;j<total\_vp;j++) //找出time的最小值

{

if(min>pl[j].time&&pl[j].pfn!=INVALID)

{

min=pl[j].time;

minj=j;

}

}

freepf\_head=&pfc[pl[minj].pfn]; //空出一个单元

pl[minj].pfn=INVALID;

pl[minj].time=0;

freepf\_head->next=NULL;

}

pl[page[i]].pfn=freepf\_head->pfn; //有空闲页面,改为有效

pl[page[i]].time=present\_time;

freepf\_head=freepf\_head->next; //减少一个free 页面

}

else

{

pl[page[i]].time=present\_time;//命中则增加该单元的访问次数

present\_time++;

}

}

printf("%6.4f",1-(float)diseffect/320);

}

**/\***

**Name:void OPT(int total\_pf)**

**Achieve:最佳置换算法（Optimal）**

**\*/**

void OPT(int total\_pf)

{

int i,j, max,maxpage,d,dist[total\_vp];

pfc\_type \*t;

initialize(total\_pf);

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) /\*页面失效\*/

{

diseffect++;

if(freepf\_head==NULL) /\*无空闲页面\*/

{

for(j=0;j<total\_vp;j++)

{

if(pl[j].pfn!=INVALID)

dist[j]=32767;

else

dist[j]=0;

}

for(j=0;j<total\_vp;j++)

{

if((pl[j].pfn!=INVALID)&&(dist[j]==32767))

{

dist[j]=j;

}

}

max=0;

for(j=0;j<total\_vp;j++)

if(max<dist[j])

{

max=dist[j];

maxpage=j;

}

freepf\_head=&pfc[pl[maxpage].pfn];

freepf\_head->next=NULL;

pl[maxpage].pfn=INVALID;

}

pl[page[i]].pfn=freepf\_head->pfn;

freepf\_head=freepf\_head->next;

}

}

printf("%6.4f",1-(float)diseffect/320);

}

**/\***

**Name:void NUR(int total\_pf )**

**Achieve:最近未使用法（No Used Recently）**

**\*/**

void NUR(int total\_pf )

{

int i,j,dp,cont\_flag,old\_dp;

pfc\_type \*t;

initialize(total\_pf);

dp=0;

for(i=0;i<total\_instruction;i++)

{

if (pl[page[i]].pfn==INVALID)//页面失效

{

diseffect++;

if(freepf\_head==NULL)//无空闲页面

{

cont\_flag=TRUE;

old\_dp=dp;

while(cont\_flag)

{

if(pl[dp].counter==0&&pl[dp].pfn!=INVALID)

cont\_flag=FALSE;

else

{

dp++;

if(dp==total\_vp)

dp=0;

if(dp==old\_dp)

for(j=0;j<total\_vp;j++)

pl[j].counter=0;

}

}

freepf\_head=&pfc[pl[dp].pfn];

pl[dp].pfn=INVALID;

freepf\_head->next=NULL;

}

pl[page[i]].pfn=freepf\_head->pfn;

freepf\_head->pn=page[i];

freepf\_head=freepf\_head->next;

}

else

pl[page[i]].counter=1;

if(i%clear\_period==0)

for(j=0;j<total\_vp;j++)

pl[j].counter=0;

}

printf("%6.4f",1-(float)diseffect/320);

}

**/\***

**Name: vodi LFU(int total\_pf)**

**Achieve:最不经常使用法（Least Frequently Used）**

**\*/**

void LFU(int total\_pf)

{

int i,j,min,minpage;

pfc\_type \*t;

initialize(total\_pf);

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID)//页面失效

{

diseffect++;

if(freepf\_head==NULL)//无空闲页面

{

min=32767;

//获取counter的使用用频率最小的内存

for(j=0;j<total\_vp;j++)

{

if(min>pl[j].counter&&pl[j].pfn!=INVALID)

{

min=pl[j].counter;

minpage=j;

}

}

freepf\_head=&pfc[pl[minpage].pfn];

pl[minpage].pfn=INVALID;

pl[minpage].counter=0;

freepf\_head->next=NULL;

}

pl[page[i]].pfn=freepf\_head->pfn;//有空闲页面,改为有效

pl[page[i]].counter++;

freepf\_head=freepf\_head->next;//减少一个free 页面

}

else

{

pl[page[i]].counter;

pl[page[i]].counter=pl[page[i]].counter+1;

}

}

printf("%6.4f",1-(float)diseffect/320);

}

int main(int)

{ int S,i;

srand((int)getpid());

for(i=0;i<total\_instruction;i+=5) //产生指令队列

{

S=(int)rand()%320;

a[i]=S; //任选一指令访问点

a[i+1]=a[i]+1; //顺序执行一条指令

a[i+2]=(int)rand()%a[i+1]; //执行前地址指令m'

a[i+3]=a[i+2]+1;//顺序执行一条指令

a[i+4]=(int)rand()%(319-a[i+2]-1)+a[i+2]+2;//执行后地址指令

}

for(i=0;i<total\_instruction;i++)//将指令序列变换成页地址流

{

page[i]=a[i]/10;

offset[i]=a[i]%10;

}

Lprintf();

printf("| 对不同帧数的各种算法的命中率比较 |\n");

Lprintf();

printf("\n");

Lprintf();

printf("|Frame +FIFO +LRU +OPT +LFU +NUR |\n");

Lprintf();

for(i=4;i<=32;i++)//用户内存工作区从4个页面到32个页面

{

printf("|%-9d",i);

printf("|");

FIFO(i);

printf(" |");

LRU(i);

printf(" |");

OPT(i);

printf(" |");

LFU(i);

printf(" |");

NUR(i);

printf(" |\n");

Lprintf();

}

system("pause");

return 0;

}