实验四 存储管理

一，实验目的

在Linux操作系统上，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，并比较他们的效率。

二，实验内容

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。地址按照下述原则生成：

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：

1. 从地址0开始：
2. 若当前指令地址为m，按照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，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选取地址

3. 重复2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.设计并实现下述算法，计算访问缺页率，并对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(First 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。

三，实验原理

1. 内存访问串生成程序：

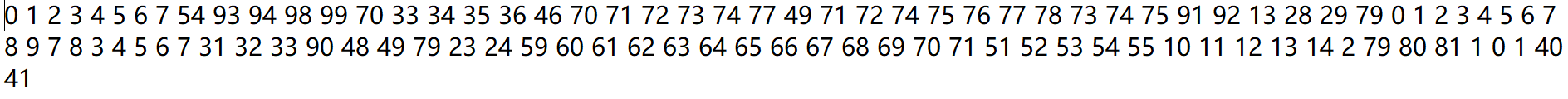
1. #include<stdio.h>

1. #include<stdlib.h>
2. #include<time.h>
3. int main()
4. {
5. FILE \*fp;
6. fp=fopen("data.txt","w");//将生成的结果写入到data.txt文件中
7. int address=0,rate,select;
8. int i;
9. srand((unsigned)time(NULL));//设置随机数种子，保证每次生成的随机数都不相同
10. fprintf(fp,"%d ",address);//向文件中写入起始地址0
11. printf("%d",address);
12. rate=rand()%10;//根据地址生成原则设置选择变量
13. for(i=1;i<100;i++)
14. {
15. printf("\nrate值为%d:\n",rate);
16. if(rate<=6 && rate>=0)//70%的概率生成的地址顺序执行
17. {
18. address=address+1;
19. fprintf(fp,"%d ",address);
20. if(address==99)//为保证地址不会超过100，当前地址为99时强制生成在前地址
21. rate=7;
22. else
23. rate=rand()%10;
24. printf("%d",address);
25. }
26. else if(rate==7)//10%的概率生成的地址在前执行
27. {
28. select=rand()%address;
29. address=select;
30. fprintf(fp,"%d ",address);
31. printf("%d",address);
32. rate=rand()%10;
33. }
34. Else//20%概率生成的地址在后执行
35. {
36. select=rand()%(99-address);
37. select=select+address+1;
38. address=select;
39. fprintf(fp,"%d ",address);
40. printf("%d",address);
41. if(address==99)
42. rate=7;
43. else
44. rate=rand()%10;
45. }
46. }
47. return 0;
48. }

说明：

因为要保证所有的地址都在[0,99]之间，所以有时会强制进行地址的向前寻址，但至少小部分事件，不会影响总体的地址生成原则。

1. 生成的内存访问串序列：



1. 页面置换算法实现程序：

#include<stdio.h>

#include<stdlib.h>

int page\_string[150];

int \*frame;

int frameNum;

struct node{

int useful[100];

int count;

};//自定义数据结构，用于记录某一个页面在哪些时刻进行访问

struct node \* number\_of\_page;//用数组的序号表示页面号

int findMax(int \*array,int arrayNum)//最近最少访问的页面数组值最大

{

int max=-2;

int maxpos=0;

int i;

for(i=0;i<arrayNum;i++)

{

if(max<array[i])

{

max=array[i];

maxpos=i;

}

}

return maxpos;

}

int isExit(int num,int \*array,int arrayNum)

{

int i;

for(i=0;i<arrayNum;i++)

{

if(array[i]==num)

return i;

}

return -1;

}

//判断要访问的页面是否在当前的内存中，如果是返回帧号，否则返回-1

void FIFO()//先进先出算法

{

int ptr=0;

int fault=0;

int i,j;

int line=0;

for(i=0;i<100;i++)

{

if(isExit(page\_string[i],frame,frameNum)<0)//如果不存在，缺页次数+1，替换指针指向的页面

{

fault++;

frame[ptr]=page\_string[i];

ptr=(ptr+1)%frameNum;//指针循环移动，模拟先进先出队列

}

printf("line=%d\t\t",i);//输出当前内存状态

for(j=0;j<frameNum;j++)

{

printf("%d\t",frame[j]);

}

printf("\tNow Referring:%d\n",page\_string[i]);

}

printf("Total Fault:%d",fault);

}

void LRU()//最近最少使用算法

{

int line=0;

int fault=0;

int \*referTimes=(int \*)malloc(sizeof(int)\*frameNum);

int i,j;

for(i=0;i<frameNum;i++)

{

referTimes[i]=0;

}

for(i=0;i<100;i++)

{

int pos=isExit(page\_string[i],frame,frameNum);//在内存中查找当前所要访问的页面

if(pos<0)//如果页面不存在，缺页次数++，替换最近最少使用的页面，然后设置新页面的计数数组为-1

{

fault++;

pos=findMax(referTimes,frameNum);

frame[pos]=page\_string[i];

}

referTimes[pos]=-1;//如果内存中存在要访问的页面，相应页面的计数数组设为-1

for(j=0;j<frameNum;j++)//所有内存中页面的计数数组每次循环都++

{

referTimes[j]++;

}

printf("line=%d\t\t",i);//输出当前内存状态

for(j=0;j<frameNum;j++)

{

printf("%d\t",frame[j]);

}

printf("\tNow Referring:%d\n",page\_string[i]);

}

printf("Total Fault:%d",fault);

}

int Opt\_max(int \* array,struct node \* a,int arrayNum,int current)//最优置换算法寻找最近最少访问页面函数

{

int i,max=-2,pos=0,j;

for(i=0;i<arrayNum;i++)

{

if(array[i]==-1)//如果内存中有空闲帧，直接返回

return i;

j=0;

for(j=0;j<a[array[i]].count;j++)

{

if(a[array[i]].useful[j]>current)//判断页面位置是否超过了当前位置，否则没有意义

{

if(a[array[i]].useful[j]>max)//找到未来最远访问的页面

{

max=a[array[i]].useful[j];

pos=i;

}

break;

}

}

}

return pos;

}

void Opt(struct node \* array)//最优置换算法

{

int i,j;

int fault=0;

for(i=0;i<100;i++)

{

int pos=isExit(page\_string[i],frame,frameNum);

if(pos<0)

{

fault++;

pos=Opt\_max(frame,array,frameNum,i);

frame[pos]=page\_string[i];

}

printf("line=%d\t\t",i);

for(j=0;j<frameNum;j++)

{

printf("%d\t",frame[j]);

}

printf("\tNow Referring:%d\n",page\_string[i]);

}

printf("Total Fault:%d",fault);

}

int main()

{

FILE \*fp;

fp=fopen("data.txt","r");

int i,j;

int page;

int per\_page;

int pageNum;

printf("Please input the number of instruction per page:");//自定义每页存放多少指令

scanf("%d",&per\_page);

pageNum=100/per\_page;

number\_of\_page=(struct node \*)malloc(sizeof(struct node)\*pageNum);

for(j=0;j<pageNum;j++)

number\_of\_page[j].count=0;

for(i=0;i<100;i++)

{

fscanf(fp,"%d",&page);

page=page/per\_page;

page\_string[i]=page;

number\_of\_page[page].useful[number\_of\_page[page].count]=i;

number\_of\_page[page].count++;

}

for(i=0;i<pageNum;i++)

{

number\_of\_page[i].useful[number\_of\_page[i].count]=100;

number\_of\_page[i].count++;

}

printf("Please input the total number of the frame:");//自定义内存中的空闲帧数目

scanf("%d",&frameNum);

frame=(int \*)malloc(sizeof(int)\*frameNum);

for(i=0;i<frameNum;i++)

{

frame[i]=-1;

}

printf("FIFO\n");

FIFO();

for(i=0;i<frameNum;i++)

{

frame[i]=-1;

}

printf("\n\nLRU\n");

LRU();

for(i=0;i<frameNum;i++)

{

frame[i]=-1;

}

printf("\n\nOpt\n");

Opt(number\_of\_page);

}

1. 程序运行结果分析：

|  |  |  |  |
| --- | --- | --- | --- |
| 内存块数量\算法类型 | FIFO | LRU | Opt |
| 4 | 40 | 39 | 34 |
| 7 | 37 | 37 | 27 |
| 10 | 32 | 33 | 24 |
| 15 | 27 | 27 | 23 |

可以发现，对于任何一种算法，随着内存块的数量的增加，缺页率有着明显的下降；同时，在其他条件相同的情况下，最优置换算法的缺页率明显小于另外两种算法，FIFO算法和LRU算法在当前的测试数据下没有显著的差异，但是经过大量的数据测试之后发现，LRU算法还是略优于FIFO算法的。

四，实验体会：

通过这次实验，我自己动手模拟了三种页面置换的算法，对三种算法的实现机制有个跟清晰的理解，同时在编程实现的过程中对三种算法的实现难度，最终的缺页率，以及算法的运行效率等等方面有了更深刻的认识，为之后的学习提供了有效的帮助。