实验报告

课程名称：操作系统

任课教师：何永忠

学生姓名：张云肖

学生学号：16281244

专业年级：安全 1601 班

学院名称：计算机学院

2018 年 5 月 16 日

实验名称:页面置换算法

班级：安全 1601　　姓名：张云肖　　学号：16281244

1. 实验目的

1. 设计和实现最佳置换算法、先进先出置换算法、最近最久未使用置换算法、页面缓冲置换算法、改进的 clock 算法
2. 通过页面访问序列随机发生器实现对上述算法的测试及性能比较。

2. 概要设计

2.1. 总体设计

当出现缺页异常，需调入新页面而内存已满时，通过页面置换算法选择能够选择被置换的物理页面，从而达到解决上述问题。
页面置换算法的设计目标是尽可能减少页面的调入调出次数，把未来不再访问或短期内不访问的页面调出。

本实验主要功能包括生成页面访问随机序列和页面置换，并输出各个算法的缺页率、算法开销。页面置换算法。其中页面置换可选择最佳置换算法、先进先出置换算法、最近最久未使用置换算法、页面缓冲置换算法、改进的 clock 算法。

2.2. 模块设计

根据实验功能，我们将程序分为 5 个模块，主要包括选择菜单设计、虚拟页面结构结构设计、页面访问随机序列设计、5 种页面置换算法设计、性能分析比较设计。

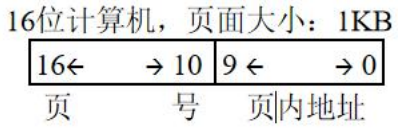
2.2.1. 选择菜单设计

主函数的功能主要是选择置换算法。能够选择使用最佳置换算法、先进先出算法、最近最久未使用算法、改进型 Clock 算法、页面缓冲置换算法或者退出。

2.2.2. 页面访问随机序列生成设计

1. 模块思路

课程前提假设：模拟的虚拟内存的地址为 16 位，页面大小为 1K；模拟的物理内存为 32K。



由于虚拟内存的地址为 16 位，则系统为进程分配的虚拟地址空间大小为【2^16B】;根据页面大小为 1K，得到物理块数=【2^16/2^10=2^6 块】，因此需要页号至少为 6 位。页号需要【6 位】，因此页内偏址需要【16-6=10】位。

2. 模块代码实现

```
struct PageInfo          //页面信息结构
{
    int   serial[64];   // 模拟的最大访问页面数，16 位信息，前 6 位表示页号，后 10 位表示页内地址
    int   flag;           // 标志位，0 表示无页面访问数据
    int   diseffect;     // 统计信息：缺页次数
    int   total_pf;      // 分配的页框数
    int   total_pn;      // 访问页面序列长度
} pf_info;
```

2.2.3. 页面访问随机序列设计

1. 模块思路

1) 确定随机生成访问序列的长度 pf_info.total_pn、工作集的起始位置 p、工作集中页面数 e、工作集移动率 m；
2) 生成 m 个取值范围在 p~p+e 之间的随机数，并存放入页面信息的数据中；
3) 生成随机数 r，如果 r>t，则为 p 生成一个新值，否则 p=p+1；
4) 如果生成序列数小于 pf_info.total_pn，则返回第二步。

2. 模块代码实现

```
void createps(void)   //  随机生成访问序列
{
        int pn;
        initialize();        //初始化相关数据结构
        cout<<"请输入要随机生成访问序列的长度";    //自定义随机生成访问序列的长度
        scanf("%d", &pn);
        int p=1, e,m;
        double t = 0.5;
        cout << "请输入工作集中页面数:";
        cin >> e;
        cout << "请输入工作集移动率:" ;
        cin >> m;
        srand((unsigned)time(NULL));     //初始化随机数队列的"种子"
        pf_info.total_pn = pn;
        for (int j = 0; j < pn;){
                for (int i = 0; i<m; i++,j++)       //产生随机访问序列
                {
                        pf_info.serial[j] = rand() % (e+1) + p;     //随机数的大小在 p-p+e 之间
                }
                double r = rand() / double(RAND_MAX);
                if (r>t){
                        p = rand() % ( 15- 1 + 1) + 1;//为 p 生成一个在 1-15 之间的新值
                }
                else{
                        p = p + 1;
                }
        }
}
```

3. 模块实现结果

```
请输入要随机生成访问序列的长度：20
请输入工作集中页面数:4
请输入工作集移动率:10

=============页面访问序列=============
   5   5   2   5   2   3   1   4   4   3
   4   2   6   6   2   6   2   6   2   6
```

2.2.4. 最佳置换算法

1. 算法简介

1) 基本思想：选择永不使用或是在最长时间内不再被访问（即距现在最长时间才会被访问）的页面淘汰出内存
2) 评价：理想化算法，具有最好性能（对于固定分配页面方式，本法可保证获得最低的缺页率），但实际上却难于实现，故主要用于算法评价参照。

2. 算法举例

假设请求分页存户管理中给进程分配的物理页面为：3，即分配的页框数。
生成的随机页面访问序列为：

4	3	4	2	4	3	2	3	1	4
3	3	5	5	2	4	3	2	2	4

且初始调入页框内不计入缺页。

```
访问  4 : 内存<  4        >
访问  3 : 内存<  4  3     >
访问  4 : 内存<  4  3     >
访问  2 : 内存<  4  3  2 >
访问  4 : 内存<  4  3  2 >
访问  3 : 内存<  4  3  2 >
访问  2 : 内存<  4  3  2 >
访问  3 : 内存<  4  3  2 >
访问  1 : 内存<  4  3  1 > ==>缺页
访问  4 : 内存<  4  3  1 >
访问  3 : 内存<  4  3  1 >
访问  3 : 内存<  4  3  1 >
访问  5 : 内存<  4  3  5 > ==>缺页
访问  5 : 内存<  4  3  5 >
访问  2 : 内存<  4  3  2 > ==>缺页
访问  4 : 内存<  4  3  2 >
访问  3 : 内存<  4  3  2 >
访问  2 : 内存<  4  3  2 >
访问  2 : 内存<  4  3  2 >
访问  4 : 内存<  4  3  2 >
```

3. 模块代码实现

```
void OPT()   // 最佳置换算法
{
    createps();            // 随机生成访问序列
    int n;
    int pstate;   //页面是否存在于内容中，0 不存在，1 存在
    int rpoint = 0;              // 页面替换指针
    int count = 0;               // 是否装满所有的页框
```

```
for (n = 0; n<pf_info.total_pf; n++)
{
    pageframe[n] = -1;      // 清除页框信息
    pagehistory[n] = 0;     // 清除页框历史
}
inpflag = 0;      //缺页标志，0 为不缺页，1 为缺页
for (vpoint = 0; vpoint<pf_info.total_pn; vpoint++)   // 执行算法
{
    pstate = findpage(pf_info.serial[vpoint]);   //查找页面是否在内存
    if (count<pf_info.total_pf)     // 开始时不计算缺页
    {
        if (pstate == 0)     // 页不存在则装入页面
        {
            pageframe[rpoint] = pf_info.serial[vpoint]; //把要调入的页面放入一个空的页框里
            rpoint = (rpoint + 1) % pf_info.total_pf;
            count++;
        }
        inpflag = 0;
    }
    else //  正常缺页置换
    {
        if (pstate == 0)//  页不存在则置换页面
        {
            int max = 0,p=0;
            for (int i = 0; i < pf_info.total_pf; i++){
                for (int n = vpoint; n<pf_info.total_pn; n++)
                {
                    if (pageframe[i] == pf_info.serial[n] && n>max)
                    {
                        max = n;
                        p = i;
                        break;
                    }
                    else if (n == pf_info.total_pn - 1){
                        p = i;
                        i = pf_info.total_pf;
                        break;
                    }
                }
            }
            pageframe[p] = pf_info.serial[vpoint];
            pf_info.diseffect++;   // 缺页次数加 1
        }
```

```
            }
            displayinfo();      // 显示当前状态
        }
        printf("缺页率%3.1f\n", (float)(pf_info.diseffect)*100.00 / vpoint);
        return;
}
```

2.2.5. 先进先出算法

1. 算法简介

基本思想：选择最先进入内存即在内存驻留时间最久的页面换出到外存，进程已调入内存的页面按进入先后次序链接成一个队列，并设置替换指针以指向最老页面
评价：简单直观，但不符合进程实际运行规律，性能较差，故实际应用极少

2. 算法举例

假设请求分页存户管理中给进程分配的物理页面为：3，即分配的页框数。
==========页面访问序列==========

1	3	1	3	2	3	2	4	4	1
3	4	2	4	3	4	3	4	2	5

且初始调入页框内不计入缺页。

```
访问  1 : 内存<  1      >
访问  3 : 内存<  1  3    >
访问  1 : 内存<  1  3    >
访问  3 : 内存<  1  3    >
访问  2 : 内存<  1  3  2 >
访问  3 : 内存<  1  3  2 >
访问  2 : 内存<  1  3  2 >
访问  4 : 内存<  4  3  2 > ==>缺页
访问  4 : 内存<  4  3  2 >
访问  1 : 内存<  4  1  2 > ==>缺页
访问  3 : 内存<  4  1  3 > ==>缺页
访问  4 : 内存<  4  1  3 >
访问  2 : 内存<  2  1  3 > ==>缺页
访问  4 : 内存<  2  4  3 > ==>缺页
访问  3 : 内存<  2  4  3 >
访问  4 : 内存<  2  4  3 >
访问  3 : 内存<  2  4  3 >
访问  4 : 内存<  2  4  3 >
访问  2 : 内存<  2  4  3 >
访问  5 : 内存<  2  4  5 > ==>缺页
```

3. 模块代码实现

```
//   FIFO 页面置换算法
void FIFO()
{
      createps();//  随机生成访问序列
      int n;
      int pstate;       //页面是否存在于内容中，0 不存在，1 存在
      int rpoint = 0;                  //  页面替换指针初始化为 0
      int count = 0;                    //  页框使用情况计数
      for (n = 0; n<pf_info.total_pf; n++) //  清除页框信息
      {
            pageframe[n] = -1;
      }
      inpflag = 0;     //缺页标志，0 为不缺页，1 为缺页
      for (vpoint = 0; vpoint<pf_info.total_pn; vpoint++)    //  执行算法
      {
            pstate = findpage(pf_info.serial[vpoint]);    //查找页面是否在内存
            if (count<pf_info.total_pf)       //  当页框未满时，不计算缺页，此时装入页面
            {
                  if (pstate == 0)       //  页不存在则装入页面
                  {
                        pageframe[rpoint] = pf_info.serial[vpoint];
                        rpoint = (rpoint + 1) % pf_info.total_pf;                  //实现模页面数，指针下移
                        count++;
                  }
                  inpflag = 0;
            }
            else          //  页面满时，正常缺页置换
            {
                  if (pstate == 0)       //页不存在则置换页面
                  {
                        pageframe[rpoint] = pf_info.serial[vpoint];
                        rpoint = (rpoint + 1) % pf_info.total_pf;
                        pf_info.diseffect++;         //  缺页次数加 1
                  }
            }
            displayinfo();            //  显示当前状态
      }       //  置换算法循环结束
      //显示缺页率
      printf("缺页率%3.1f\n", (float)(pf_info.diseffect)*100.00 / vpoint);
      return;
}
```

2.2.6. 最近最久未使用置换算法

1. 算法简介

基本思想：以“最近的过去”作为“最近的将来”的近似，选择最近一段时间最长时间未被访问的页面淘汰出内存
评价：适用于各种类型的程序，性能较好，但需要较多的硬件支持

2. 算法举例

假设请求分页存户管理中给进程分配的物理页面为：3，即分配的页框数。
==========页面访问序列==========

1	1	3	2	2	2	3	1	2	2
15	15	13	12	14	14	12	13	12	14

且初始调入页框内不计入缺页。

```
访问  1 : 内存<  1        >
访问  1 : 内存<  1        >
访问  3 : 内存<  1  3     >
访问  2 : 内存<  1  3  2  >
访问  2 : 内存<  1  3  2  >
访问  2 : 内存<  1  3  2  >
访问  3 : 内存<  1  3  2  >
访问  1 : 内存<  1  3  2  >
访问  2 : 内存<  1  3  2  >
访问  2 : 内存<  1  3  2  >
访问 15 : 内存<  1 15  2  > ==>缺页
访问 15 : 内存<  1 15  2  >
访问 13 : 内存< 13 15  2  > ==>缺页
访问 12 : 内存< 13 15 12  > ==>缺页
访问 14 : 内存< 13 14 12  > ==>缺页
访问 14 : 内存< 13 14 12  >
访问 12 : 内存< 13 14 12  >
访问 13 : 内存< 13 14 12  >
访问 12 : 内存< 13 14 12  >
访问 14 : 内存< 13 14 12  >
```

3. 模块代码实现

```
//   LRU 页面置换算法
void LRU()
{
        createps();             // 随机生成访问序列
        int n;
        int pstate;   //页面是否存在于内容中，0 不存在，1 存在
        int max;
        int rpoint = 0;                 // 页面替换指针
        int count = 0;                   // 是否装满所有的页框
        for (n = 0; n<pf_info.total_pf; n++)
```

```
    {
        pageframe[n] = -1;      // 清除页框信息
        pagehistory[n] = 0;     // 清除页框历史
    }
    inpflag = 0;       //缺页标志，0 为不缺页，1 为缺页
    for (vpoint = 0; vpoint<pf_info.total_pn; vpoint++)   // 执行算法
    {
        pstate = findpage(pf_info.serial[vpoint]);   //查找页面是否在内存
        if (count<pf_info.total_pf)     // 开始时不计算缺页
        {
            if (pstate == 0)     // 页不存在则装入页面
            {
                pageframe[rpoint] = pf_info.serial[vpoint]; //把要调入的页面放入一个空的页框里
                rpoint = (rpoint + 1) % pf_info.total_pf;
                count++;
            }
            inpflag = 0;
        }
        else // 正常缺页置换
        {
            if (pstate == 0)// 页不存在则置换页面
            {
                max = 0;
                for (n = 1; n<pf_info.total_pf; n++)
                {
                    if (pagehistory[n]>pagehistory[max])
                    {
                        max = n;
                    }
                }
                rpoint = max;
                pageframe[rpoint] = pf_info.serial[vpoint];
                pagehistory[rpoint] = 0;
                pf_info.diseffect++;   // 缺页次数加 1
            }
        }
        displayinfo();      // 显示当前状态
    }      // 置换算法循环结束
    printf("缺页率%3.1f\n", (float)(pf_info.diseffect)*100.00 / vpoint);
    return;
}
```

2.2.7. 改进的 clock 算法

1. 算法简介

基本思想：

1) 从查寻指针当前位置起扫描内存分页循环队列，选择 A=0 且 M=0 的第一个页面淘汰；若未找到，转 2)
2) 开始第二轮扫描，选择 A=0 且 M=1 的第一个页面淘汰，同时将经过的所有页面访问位置 0；若不能找到，转 1)

评价：与简单 Clock 算法相比，可减少磁盘的 I/O 操作次数，但淘汰页的选择可能经历多次扫描，故实现算法自身的开销增大

2. 算法举例

假设请求分页存户管理中给进程分配的物理页面为：3，即分配的页框数。

==========页面访问序列==========

3 4 2 2 3 3 3 1 3 4

2 4 4 4 5 3 5 5 4 5

且初始调入页框内不计入缺页。

```
访问  3 : 内存<  3        >
访问  4 : 内存<  3  4     >
访问  2 : 内存<  3  4  2 >
访问  2 : 内存<  3  4  2 >
访问  3 : 内存<  3  4  2 >
访问  3 : 内存<  3  4  2 >
访问  3 : 内存<  3  4  2 >
访问  1 : 内存<  3  1  2 > ==>缺页
访问  3 : 内存<  3  1  2 >
访问  4 : 内存<  3  4  2 > ==>缺页
访问  2 : 内存<  3  4  2 >
访问  4 : 内存<  3  4  2 >
访问  4 : 内存<  3  4  2 >
访问  4 : 内存<  3  4  2 >
访问  5 : 内存<  5  4  2 > ==>缺页
访问  3 : 内存<  3  4  2 > ==>缺页
访问  5 : 内存<  5  4  2 > ==>缺页
访问  5 : 内存<  5  4  2 >
访问  4 : 内存<  5  4  2 >
访问  5 : 内存<  5  4  2 >
```

3. 模块代码实现

```
//改进 Clock()算法
void gClock()
{
    createps();          // 随机生成访问序列
    int n;
    int pstate;   //页面是否存在于内容中，0 不存在，1 存在
```

```
int max;
int rpoint = 0;                 // 页面替换指针
int count = 0;                  // 是否装满所有的页框
int paccess[10];  //记录内存框中访问位
int pmodify[10];       //记录内存框中修改位
srand((unsigned)time(NULL));     //初始化随机数队列的"种子"
for (n = 0; n<pf_info.total_pf; n++)
{
        pageframe[n] = -1;       // 清除页框信息
        paccess[n] = 0;
        pmodify[n] = rand() % 2 ;     //随机数 0 或 1
}
inpflag = 0;       //缺页标志，0 为不缺页，1 为缺页
for (vpoint = 0; vpoint<pf_info.total_pn; vpoint++)   // 执行算法
{
        pstate = findpage(pf_info.serial[vpoint]);   //查找页面是否在内存
        for (n = 0; n<pf_info.total_pf; n++)
        {
                if (pageframe[n] == pf_info.serial[vpoint])
                {
                        paccess[n] = 1;
                }
        }
        if (count<pf_info.total_pf)     // 开始时不计算缺页
        {
                if (pstate == 0)     // 页不存在则装入页面
                {
                        pageframe[rpoint] = pf_info.serial[vpoint]; //把要调入的页面放入一个空的页框里
                        rpoint = (rpoint + 1) % pf_info.total_pf;
                        count++;
                }
                inpflag = 0;
        }
        else //  正常缺页置换
        {
                if (pstate == 0)//  页不存在则置换页面
                {
                        rpoint = 0;
                        int flag=0;
                        for (n = 0; n<pf_info.total_pf; n++)
                        {
                                if (paccess[n] == 0 && pmodify[n]==0)
                                {
                                        rpoint = n;
```

```
                    break;
                }
                if (n == pf_info.total_pf - 1)flag = 1;
            }
            if (flag){
                for (n = 0; n<pf_info.total_pf; n++)
                {
                    if (paccess[n] == 0 && pmodify[n] == 1)
                    {
                        rpoint = n;
                        break;
                    }
                    paccess[n] = 0;
                }
            }
            pageframe[rpoint] = pf_info.serial[vpoint];
            pmodify[rpoint] = rand() % 2;              //随机植入修改位值
            pf_info.diseffect++;   // 缺页次数加 1
        }
    }
    displayinfo();      // 显示当前状态
}   // 置换算法循环结束
printf("缺页率%3.1f\n", (float)(pf_info.diseffect)*100.00 / vpoint);
return;
}
```

2.2.8. 页面缓冲置换算法

1. 算法简介

基本思想：

1) 设立空闲页面链表和已修改页面链表

2) 采用可变分配和基于先进先出的局部置换策略，并规定被淘汰页先不做物理移动，而是依据是否修改分别挂到空闲页面链表或已修改页面链表的末尾

3) 空闲页面链表同时用于物理块分配

4) 当已修改页面链表达到一定长度如 Z 个页面时，一起将所有已修改页面写回磁盘，故可显著减少磁盘 I/O 操作次数

2. 算法举例

假设请求分页存户管理中给进程分配的物理页面为：3，即分配的页框数。

==========页面访问序列==========

2 2 4 2 4 1 2 4 3 4

15 15 14 15 13 15 14 13 15 14

且初始调入页框内不计入缺页。

```
访问  2 : 内存<  2           >
访问  2 : 内存<  2           >
访问  4 : 内存<  2  4        >
访问  2 : 内存<  2  4        >
访问  4 : 内存<  2  4        >
访问  1 : 内存<  2  4  1     >
访问  2 : 内存<  2  4  1     >
访问  4 : 内存<  2  4  1     >
访问  3 : 内存<     4  1  3  > ==>缺页
访问  4 : 内存<     4  1  3  >
访问 15 : 内存<        1  3 15 > ==>缺页
访问 15 : 内存<        1  3 15 >
访问 14 : 内存< 14        3 15 > ==>缺页
访问 15 : 内存< 14        3 15 >
访问 13 : 内存<    13     3 15 > ==>缺页
访问 15 : 内存<    13     3 15 >
访问 14 : 内存< 14        3 15 > ==>缺页
访问 13 : 内存<    13     3 15 > ==>缺页
访问 15 : 内存<    13     3 15 >
访问 14 : 内存< 14        3 15 > ==>缺页
```

3. 模块代码实现

```
//PBA()算法
void PBA(){
	struct Pb
	{
		int Memnum;			//空闲列对应内存块号
		int Pagenum;		//空闲列对应访问页号
	}pb[2];
	createps();		// 随机生成访问序列
	int n;
	int pstate;   //页面是否存在于内容中，0 不存在，1 存在
	int rpoint = 0;		// 页面替换指针
	int count = 0;		// 是否装满所有的页框
	for (n = 0; n < pf_info.total_pf; n++)
	{
		pageframe[n] = -1;	// 清除页框信息
		existence[n] = 1;//页框存在为置 1
	}
	existence[pf_info.total_pf] = 0;
	existence[pf_info.total_pf+1] = 0;//设置两位的存在位
	pb[0].Memnum = pf_info.total_pf;
	pb[0].Pagenum = -1;
	pb[1].Memnum = pf_info.total_pf+1;
	pb[1].Pagenum = -1;			//初始化空闲链
	inpflag = 0;	//缺页标志，0 为不缺页，1 为缺页
```

```
for (vpoint = 0; vpoint<pf_info.total_pn; vpoint++)   // 执行算法
{
    pstate = 0;//默认缺页
    for (n = 0; n<pf_info.total_pf+2; n++)
    {
        if (pageframe[n] == pf_info.serial[vpoint] && existence[n]==1)
        {
            pstate = 1;//访问页面存在于内存中
            inpflag = 0;
            break;
        }
    }
    if (count<pf_info.total_pf)    // 开始时不计算缺页
    {
        if (pstate == 0)    // 页不存在则装入页面
        {
            pageframe[rpoint] = pf_info.serial[vpoint]; //把要调入的页面放入一个空的页框里
            rpoint = (rpoint + 1) % pf_info.total_pf;
            existence[rpoint] = 1;
            count++;
        }
        inpflag = 0;
    }
    else // 正常缺页置换
    {
        if (pstate == 0)// 页不存在则置换页面
        {
            inpflag = 1;
            int flag = 0;//页面是否在空闲链内,0 不在，1 在
            for (int i = 0; i < 2; i++){
                if (pb[i].Pagenum == pf_info.serial[vpoint]){
                    existence[pb[i].Memnum] = 1;
                    pageframe[pb[i].Memnum] = pb[i].Pagenum;
                    if (i == 0){//如果是链首弹出
                        pb[0].Memnum = pb[1].Memnum;
                        pb[0].Pagenum = pb[1].Pagenum; //空闲链首移动一位
                        existence[rpoint] = 0;//从访问页框根据先进先出弹出页框,放入空闲链尾
                        pb[1].Memnum = rpoint;
                        pb[1].Pagenum = pageframe[rpoint];
                    }
                    else{//如果是链尾
                        existence[rpoint] = 0;  //从访问页框根据先进先出弹出页框,放入空闲
                        pb[1].Memnum = rpoint;
                        pb[1].Pagenum = pageframe[rpoint];
```

```
                                }
                                flag = 1;
                            }
                        }
                        if (flag == 0){
                            existence[pb[0].Memnum] = 1;
                            pageframe[pb[0].Memnum] = pf_info.serial[vpoint];
                            pb[0].Memnum = pb[1].Memnum;
                            pb[0].Pagenum = pb[1].Pagenum; //空闲链首移动一位
                            existence[rpoint] = 0;          //从访问页框根据先进先出弹出页框,放入空闲链尾
                            pb[1].Memnum = rpoint;
                            pb[1].Pagenum = pageframe[rpoint];
                        }
                        rpoint = (rpoint + 1) % pf_info.total_pf;//页面指针下移
                        while (existence[rpoint] == 0)
                            rpoint = (rpoint + 1) % pf_info.total_pf;//页面指针下移
                        pf_info.diseffect++;        // 缺页次数加 1
                    }
                }
                displayinfoPBA();       // 显示当前状态
            }       // 置换算法循环结束
            printf("缺页率%3.1f\n", (float)(pf_info.diseffect)*100.00 / vpoint);
            return;
        }
```

2.2.9. 性能分析比较设计

假设随机序列生成长度为 20，测试得如下结果

算法缺页率					
页框数	OPT	FIFO	LRU	gClock	PBA
3	5%	10%	10%	10%	10%
4	5%	15%	5%	10%	20%
5	5%	10%	10%	5%	10%
6	0%	5%	10%	0%	5%
平均值	3.75%	10.00%	8.75%	6.25%	11.25%

得：算法缺页率为：PBA>FIFO>LRU>gClock>OPT

算法开销/时间					
页框数	OPT	FIFO	LRU	gClock	PBA
3	9.854	3.242	8.85	5.016	3.586
4	6.462	3.966	3.422	5.205	4.877
5	9.157	4.282	5.932	3.705	4.933

6	9.144	5.673	5.9	4.863	5.503
平均值	8.65425	4.29075	6.026	4.69725	4.69725

得算法时间开销为:OPT>LRU>gClock>PBA>FIFO