

**操作系统与安全**

**课程设计报告**

**学院名称 信息科学与技术学院**

**专业名称 网络空间安全**

**学生姓名 彭重严**

**学生学号 201817030115**

**指导教师 杨国城**

**实验地点 7203**

**2020年 12 月**

目录

[设计项目一：作业调度算法的模拟实现 1](#_Toc61820478)

[一、实现目的 1](#_Toc61820479)

[二、实验准备 1](#_Toc61820480)

[三、实验基本知识及原理 1](#_Toc61820481)

[四、实验说明 2](#_Toc61820482)

[五、实验内容 3](#_Toc61820483)

[六、完整代码 4](#_Toc61820484)

[七、实验结果 9](#_Toc61820485)

[八、实验总结分析 10](#_Toc61820486)

[设计项目二：页面调度算法的模拟实现 11](#_Toc61820487)

[一、实现目的 11](#_Toc61820488)

[二、实验准备 11](#_Toc61820489)

[三、实验基本知识及原理 11](#_Toc61820490)

[四、实验说明 12](#_Toc61820491)

[五、实验内容 13](#_Toc61820492)

[六、完整代码 16](#_Toc61820493)

[七、实验结果（附上运行结果的截图） 23](#_Toc61820494)

[八、实验总结分析 23](#_Toc61820495)

[实验心得及教师评语 24](#_Toc61820496)

# 设计项目一：作业调度算法的模拟实现

## 一、实现目的

1）掌握周转时间、等待时间、平均周转时间等概念及其计算方法。

2）理解五种常用的作业调度算法（FCFS、SJF、HRRF、HPF、RR），区分算法之间的差异性，并用C语言模拟实现各算法。

3）了解操作系统中高级调度、中级调度和低级调度的区别与联系。

## 二、实验准备

1）掌握程序、进程、作业的基本概念。

2）掌握进程调度、作业调度的区别和联系。

3）掌握C语言基本语法和struct结构及其用法。

## 三、实验基本知识及原理

1.基本概念

程序：程序是指静态的指令集合，它不占用系统的运行资源，可以长久地保存在磁盘中。

进程：进程是指进程实体（由程序、数据和进程控制块构成）的运行过程，是系统进行资源分配和调度的一个独立单位。进程执行程序，但进程与程序之间不是一一对应的。通过多次运行，一个程序可以包含多个进程；通过调用关系，同一进程可以被多个程序包含（如一个DLL文件可以被多个程序运用）。

作业：作业由一组统一管理和操作的进程集合构成，是用户要求计算机系统完成的一项相对独立的工作。作业可以是完成了编译、链接之后的一个用户程序，也可以是各种命令构成的一个脚本。

作业调度：作业调度是在资源满足的条件下，将处于就绪状态的作业调入内存，同时生成与作业相对应的进程，并为这些进程提供所需要的资源。作业调度适用于多道批处理系统中的批处理作业。根据作业控制块中的信息，检查系统是否满足作业的资源要求，只有在满足作业调度的资源需求的情况下，系统才能进行作业调度。

2.基本调度算法

（1)先来先服务(First-Come First-Served, FCFS)调度算法

先来先服务调度算法遵循按照进入后备队列的顺序进行调度的原则。该算法是一种非抢占式的算法，也是到目前为止最简单的调度算法，其编码实现非常容易。该算法仅考虑了作业到达的先后顺序，而没有考虑作业的执行时间长短、作业的运行特性和作业对资源的要求。

（2)短作业优先(Shortest-Job-First, SJF)调度算法

短作业优先调度算法根据作业控制块中指出的执行时间，选取执行时间最短的作业优先调度。本实验中规定，该算法是非抢占式的，即不允许立即抢占正在执行中的长进程，而是等当前作业执行完毕再进行调度。

（3)响应比高者优先(High-Response-Ratio-First, HRRF)调度算法

FCFS调度算法只片面地考虑了作业的进入时间，短作业优先调度算法考虑了作业的运行时间而忽略了作业的等待时间。响应比高者优先调度算法为这两种算法的折中。响应比为作业的响应时间与作业需要执行的时间之比。作业的响应时间为作业进入系统后的等待时间与作业要求处理器处理的时间之和。

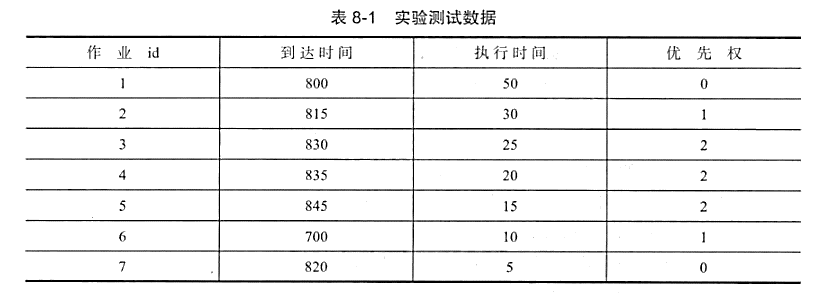
（4)优先权高者优先(Highest-Priority-First, HPF)调度算法

优先权高者优先调度算法与响应比高者优先调度算法十分相似，是根据作业的优先权进行作业调度，每次选取优先权高的作业优先调度。作业的优先权通常用一个整数表示，也叫优先数。优先数的大小与优先权的关系由系统或者用户规定。优先权高者优先调度算法综合考虑了作业执行时间和等待时间的长短、作业的缓急度、作业对外部设备的使用情况等因素，根据系统设计目标和运行环境而给定各个作业的优先权，决定作业调度的先后顺序。

## 四、实验说明

1）本实验所选用的调度算法均默认为非抢占式

2）实验所用的测试数据如表8-1所示



3）本实验设计的作业的数据结构：

typedef struct node

{

int number; //作业号

int reach\_time; //作业抵达时间

int need\_time; //作业的执行时间

int privilege; //作业优先权

float excellent; //响应比

int start time; //作业开始时间

1nt wait\_time; //等待时间

int visited; //作业是否被访问过

bool 1sreached; //作业是否已经抵达

}Job;

4）重要函数说明：

void initial jobso：初始化所有作业信息。void resetjinfo0 重置所有作业信息。

int findminjob（job jobs J，int count）：找到执行时间最短的作业。输入参数为所有的作业信息及待查找的作业总数，输出为执行时间最短的作业id.

int findrearlyjob（job jobs.int count）：找到到达最早的作业。输入参数为所有的作业信息及待查找的作业总数，输出参数为最早到达的作业id.

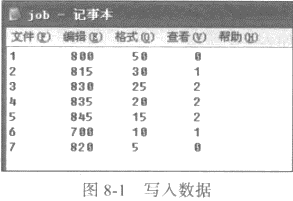
void readJobdata）：读取作业的基本信息。void FCFSO先来先服务算法

void SFJschdulejobjob jobsl]，int count）：短作业优先算法。输入参数为所有的作业信息及待查找的作业总数。

## 五、实验内容

下面我们设计算法来模拟作业调度。

1）将表8—1中的数据去掉第一行写入文本文件中，每行之间用换行符分隔，每列之间用空格分隔，如图8-1所示。



2）运行本实验的参考代码，参见代码8-1.

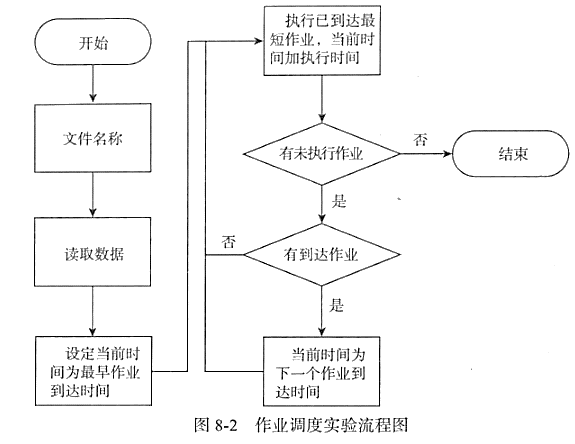
①通过程序的打印信息来检查作业信息的读入是否正确。

②运行FCFS算法，检验其运算结果是否正确。

③根据图8-2补充短作业优先代码，并计算其等待时间和周转时间。

④尝试编写时间片轮转算法和高响应比优先算法。

实验流程图如下：



## 六、完整代码

#include<stdio.h>

#include<string.h>

#include<stdbool.h>

#include<stdlib.h>

//最大作业量

#define MAXJOB 50

// 作业的数据结构

typedef struct node

{

int number; // 作业号

int reach\_time; // 作业抵达时间

int need\_time; // 作业的执行时间

int privilege; // 作业优先权

float excellent; // 响应比

int start\_time; // 作业开始时间

int wait\_time; // 等待时间

int visited; // 作业是否被访问过

bool isreached; // 作业是否抵达

}job;

job jobs[MAXJOB]; // 作业序列

int quantity; // 作业数量

// 初始化作业序列

void initial\_jobs()

{

int i;

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

{

jobs[i].number = 0;

jobs[i].reach\_time = 0;

jobs[i].privilege=0;

jobs[i].excellent=0;

jobs[i].start\_time=0;

jobs[i].wait\_time=0;

jobs[i].visited=0;

jobs[i].isreached=false;

}

quantity = 0;

}

// 重置全部作业

void reset\_jinfo()

{

int i=0;

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

{

jobs[i].start\_time=0;

jobs[i].wait\_time=0;

jobs[i].visited=0;

}

}

// 查找当前current\_time已达到未执行的最短作业，若无返回 -1

int findminjob(job jobs[], int count)

{

int minjob=-1;

int minloc=-1;

int i=0;

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

{

if(minloc==-1)

{

if(jobs[i].isreached == true && jobs[i].visited==0)

{

minjob=jobs[i].need\_time;

minloc=i;

}

}

else if(minjob>jobs[i].need\_time && jobs[i].visited==0 && jobs[i].isreached==true)

{

minjob=jobs[i].need\_time;

minloc=1;

}

}

return minloc;

}

// 查找最早到达作业，若全部到达返回-1

int findrearlyjob(job jobs[], int count)

{

int rearlyloc=-1;

int rearlyjob=-1;

int i;

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

{

if(rearlyloc == -1)

{

if(jobs[i].visited==0)

{

rearlyloc=i;

rearlyjob=jobs[i].reach\_time;

}

}else if(rearlyloc>jobs[i].reach\_time && jobs[i].visited==0){

rearlyjob=jobs[i].reach\_time;

rearlyloc=i;

}

}

return rearlyloc;

}

// 读取作业数据

void readJobdata()

{

FILE \*fp;

char fname[20];

int i;

//输入测试文件的文件名

printf("please input job data file name: \n");

scanf("%s", fname);

if((fp=fopen(fname, "r")) == NULL)

{

printf("Error, open file failed, please check filename\n");

}else{

//依次读取作业信息

while(!feof(fp))

{

if(fscanf(fp, "%d %d %d %d", &jobs[quantity].number, &jobs[quantity].reach\_time, &jobs[quantity].need\_time, &jobs[quantity].privilege)==4)

quantity++;

}

//打印作业信息

printf("output the origin job data\n");

printf("----------------------------------\n\n");

printf("\tjobId\treachTime\tneedTime\tprivilege\n");

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

{

printf("\t%-8d\t%-8d\t%-8d\t%-8d\n", jobs[quantity].number, jobs[quantity].reach\_time, jobs[quantity].need\_time, jobs[quantity].privilege);

}

}

}

// FCFS

void FCFS()

{

int i;

int current\_time=0;

int loc;

int total\_waittime=0;

int total\_roundtime=0;

//获取最近到达的作业

loc = findrearlyjob(jobs, quantity);

//输出作业流

printf("\n\nFCFS算法作业流n");

printf("-------------------------\n");

printf("\tjobId\treachTime\tstartTime\twaitTime\troundTime\n");

current\_time = jobs[loc].reach\_time;

//每次循环找出最先到达的作业并打印相关信息

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

{

if(jobs[loc].start\_time>current\_time)

{

jobs[loc].start\_time=jobs[loc].reach\_time;

current\_time=jobs[loc].reach\_time;

}

else{

jobs[loc].start\_time=current\_time;

}

jobs[loc].wait\_time=current\_time - jobs[loc].need\_time;

printf("\t%-8d\t%-8d\t%-8d\t%-8d\t%-8d\n", loc+1,

jobs[loc].reach\_time, jobs[loc].start\_time, jobs[loc].wait\_time,

jobs[loc].wait\_time+jobs[loc].need\_time

);

jobs[loc].visited=1;

current\_time += jobs[loc].need\_time;

total\_waittime += jobs[loc].wait\_time;

total\_roundtime = total\_roundtime + jobs[loc].wait\_time+jobs[loc].need\_time;

//获取剩余作业中最近到达作业

loc=findrearlyjob(jobs, quantity);

}

printf(" 总等待时间： %-8d 总周转时间 %-8d\n", total\_waittime, total\_roundtime);

printf(" 平均等待时间：%4.2f 平均周转时间：%4.2f\n", (float)total\_waittime/(quantity), (float)total\_roundtime/(quantity));

}

//短作业优先作业调度

void SFJschdulejob(job jobs[], int count)

{

}

int main()

{

initial\_jobs();

readJobdata();

FCFS();

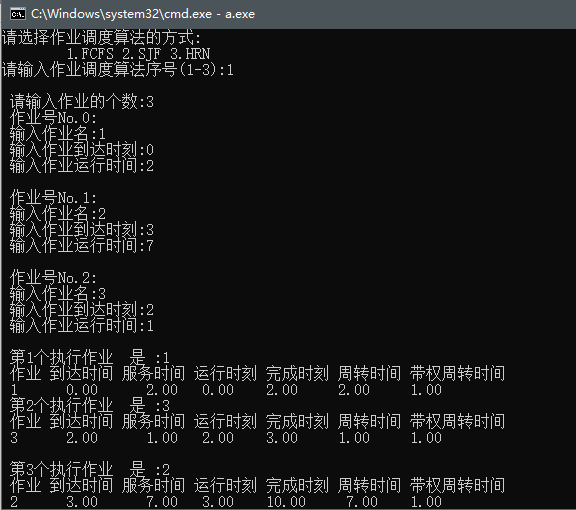
return 0;

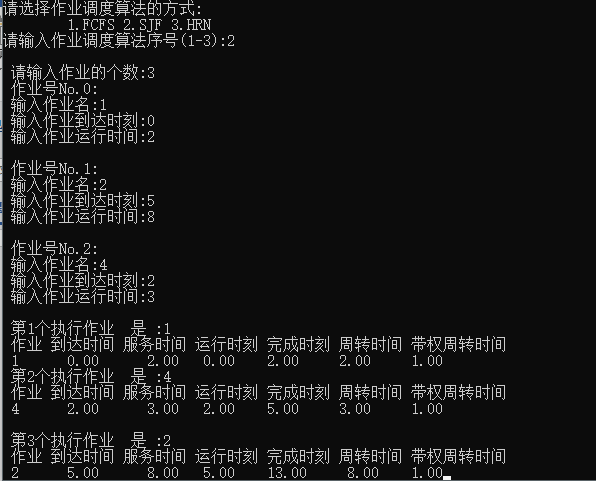
}

return 0;

}

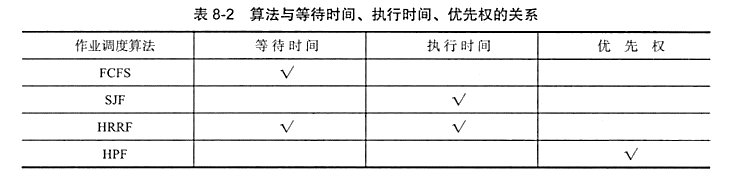
## 七、实验结果





## 八、实验总结分析

从四种算法的测试数据来看，算法思想尽不同，所需的等待时间和周转时间也不同。



由表8—2得出FCFS算法仅考虑了作业的等待时间，等待时间长的优先考虑； SJF算法主要考虑作业的执行时间，执行时间短的优先考虑；HRRF算法同时考虑了作业的等待时间和执行时间，是FCFS和SJF算法的折中；HPF算法仅考虑作业的优先权，优先权高者先执行。

从实验结果中可以发现，对测试数据而言，并非HRRF算法的平均等待时间和平均周转时间最短。对于这组作业， SJF算法的平均等待时间和平均周转时间比HRRF算法和HPF算法的短，说明最适合这个作业的调度算法是SJF。

由此可以得出，算法的好坏与具体的作业有关。如果对于a作业， A算法的平均等待时间和周转时间是最短的，那说明A算法是最适合a作业的调度算法。

# 设计项目二：页面调度算法的模拟实现

## 一、实现目的

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）先进先出算法(FIFO);

2）最近最少使用算法(LRU);

3）最佳淘汰算法(OPT);

4）最少访问页面算法(LFU);

5）最近最不经常使用算法(NUR)。

命中率计算公式如下:

命中率=(1-页面失效次数)/页地址流长度

## 二、实验准备

1）虚拟机Vmware 15.5.4软件

2）Ubuntu系统

3）相关编译环境

## 三、实验基本知识及原理

1）先进先出算法(FIFO)

最简单的分页替换算法就是先进先出算法，当每次有新的分页需要调入时，会选择调入内存时间最久的分页换出。有两种实现的方法：第一种是记录每个分页被调入到页框的时间，当每次需要换出分页时，会找到调入时间最早的一页，也就是在主存储器中存在最久的分页。另外一种方式就是利用FIFO队列来实现，当要进行分页替换时，就把队列最前端的分页换出，再把要调入的分页放到队列的末端。

2）最近最少使用算法(LRU)

LRU是Least Recently Used的缩写，即最近最少使用，是一种常用的页面置换算法，选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间t，当须淘汰一个页面时，选择现有页面中其t值最大的，即最近最少使用的页面予以淘汰。

3）最佳淘汰算法(OPT)

最佳淘汰算法(OPT)，淘汰以后不再需要的或者在最长时间以后才会用到的页面，这一算法不可能实现，但它可以作为衡量其他页面淘汰算法优劣的一个标准。

4）最少访问页面算法(LFU)

如果一个数据在最近一段时间很少被访问到，那么可以认为在将来它被访问的可能性也很小。因此，当空间满时，最小频率访问的数据最先被淘汰。

5）最近最不经常使用算法(NUR)

如果一个数据在最近一段时间没有被访问到，那么可以认为在将来它被访问的可能性也很小。因此，当空间满时，没被访问的数据最先被淘汰。

## 四、实验说明

本实验的程序设计基本上按照实验内容进行，即首先用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，相关定义结构如下：

1）数据结构

①页面类型

typedef struct

{

int pn,pfn,counter,time;

}pl\_type;

其中pn为页号，pfn为面号，counter为一个周期内访问该页面的次数，time为访问时间。

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{total\_vp}, \*freef\_head, \*busypf\_head;

pfc\_type \*buspf\_tail;

其中pfc[total\_vp]定义用户进程虚页控制结构；

freepf\_head为空页面头的指针

busypf\_head为忙页面头的指针

busypf\_tail为忙页面尾的指针

2）函数定义

void initialize()：初始化函数，给每个相关的页面赋值

void FIFO()：计算使用FIFO算法时的命中率

void LRU()：计算使用LRU算法时的命中率

void OPT()：计算使用OPT算法时的命中率

void LFU()：计算使用LFU算法时的命中率

void NUR()：计算使用NUR算法时的命中率

3）变量定义

int a[total\_instruction]：指令流数据组

int page[total\_instruction]：每条指令所属页号

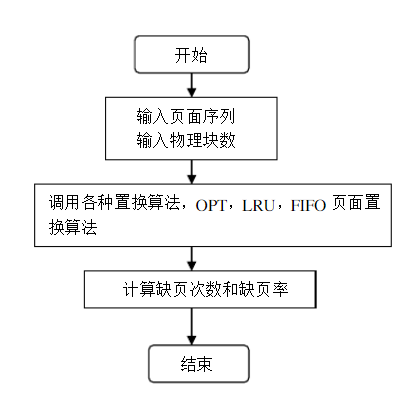
int offset[total\_instruction]：每页装入10条指令后取模运算页号偏移

int total\_pf：用户进程的内存页面数

int diseffect：页面失效次数

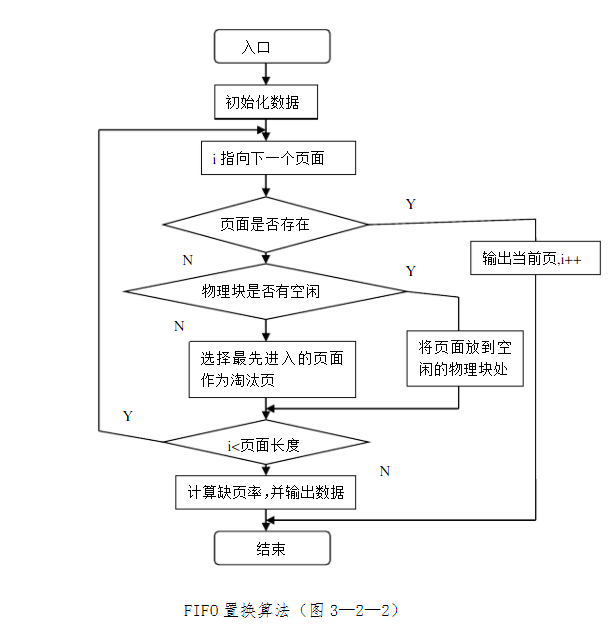
## 五、实验内容

**概要设计：**

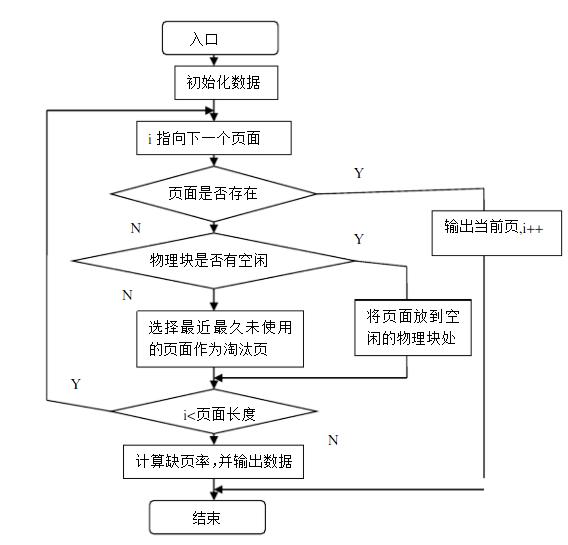


**各个算法的实现流程图：**

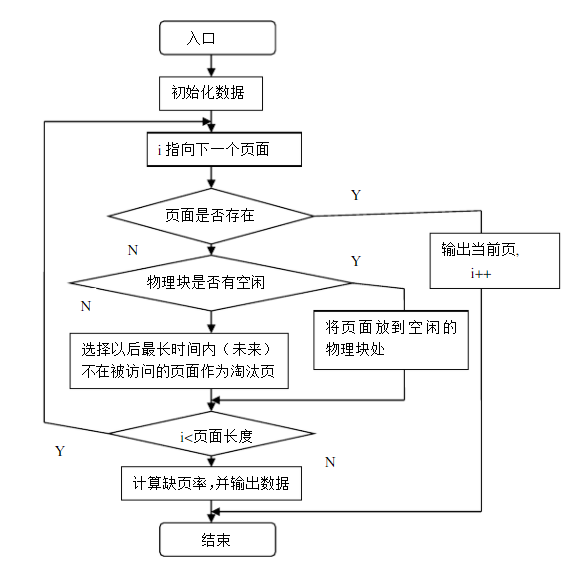
**1）先来先服务算法(FIFO)**



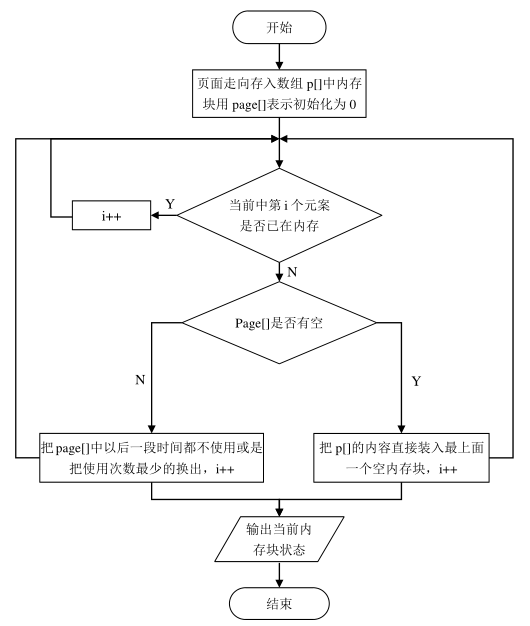
**2）最近最少使用算法(LRU)**



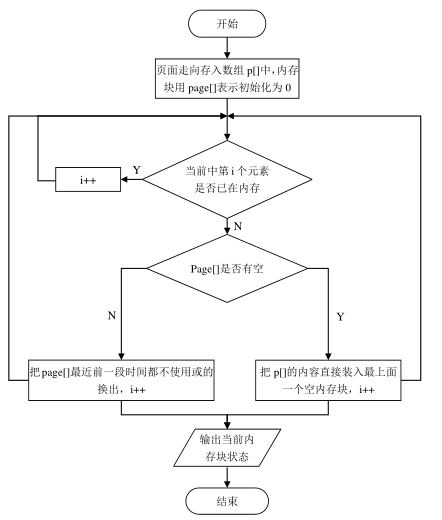
**3）最佳淘汰算法(OPT)**



**4）最少访问页面算法(LFU)**



**5）最近最不经常使用算法(NUR)**



## 六、完整代码

#include<stdlib.h>

#include<stdio.h>

#define TRUE 1

#define FALSE 0

#define INVALID -1

#define NULL 0

#define total\_instruction 320

#define total\_vp 32

#define clear\_period 50

typedef struct

{

int pn, pfn, counter, time;

} pl\_type;

pl\_type pl[32];

typedef struct pfc\_struct

{

int pn, pfn;

struct pfc\_struct \*next;

}pfc\_type;

pfc\_type pfc[32], \*freepf\_head, \*busypf\_head, \*busypf\_tail;

int diseffect, a[total\_instruction];

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

void initialize();

void FIFO();

void LRU();

void OPT();

void LFU();

void NUR();

void main(){

int s, i, j;

srand(10\*getpid()); // 每次运行时进程产生的进程号不同，所以可以用来作为种子。

s=(float)319\*rand()/32767/32767/2+1;

for(i=0;i<total\_instruction;i+=4) // 产生指令队列

{

if(s<0||s>319)

{

printf("When i==%d, Error, s==%d\n", i, s);

exit(0);

}

a[i]=s; //任选一个指令访问点

a[i+1]=a[i]+1; //顺序执行一条指令

a[i+2]=(float)a[i]\*rand()/32767/32767/2; //执行前地址指令m

a[i+3]=a[i+2]+1; //顺序执行一条指令

s = (float)(318-a[i+2]) \* rand()/32767/32767/2+a[i+2]+2;

if((a[i+2]>318)||(s>319))

printf("a[%d+2], a number which is: %d and s==%d\n", i, a[i+2],s);

}

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;

}

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

{

printf("%2d page frames", i);

FIFO(i);

LRU(i);

OPT(i);

LFU(i);

NUR(i);

printf("\n");

}

}

void initialize(int total\_pf) //初始化相关数据结构

{

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=0;i<total\_pf-1;i++)

{

pfc[i].next = &pfc[i+1];

pfc[i].pfn = i;

} //建立pfc[i-1]和pfc[i]之间的链接

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

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

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

}

void FIFO( total\_pf) //先进先出算法FIFO(First In First Out)

int total\_pf;

{

int i,j;

pfc\_type \* p;

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

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) //页面时效

{

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;

busypf\_head=p;

}

p=freepf\_head->next; //按FIFO方式调新页面入内存页面

freepf\_head->next=NULL;

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

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

if(busypf\_tail==NULL)

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

else

{

busypf\_tail->next=freepf\_head; //空闲页面减少一个

busypf\_tail=freepf\_head;

}

freepf\_head=p;

}

}

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

}

void LFU(int total\_pf) //LFU算法

{

int i,j,min,minpage;

pfc\_type \*t;

initialize(total\_pf);

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

{

if(pl[page[i]].pfn==INVALID)

{

diseffect++;

if(freepf\_head==NULL)

{

min = 32767;

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

{

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

{

min=pl[j].counter;

minpage=j;

}

pl[j].counter=0;

}

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

pl[minpage].pfn=INVALID;

freepf\_head->next = NULL;

}

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

freepf\_head=freepf\_head->next;

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

}

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

}

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

}

void LRU( total\_pf) //LRU算法

int total\_pf;

{

int min, minj,i,j,present\_time;

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;

for(j=0;j<total\_vp;j++) //找出最小time值

{

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

{

min=pl[j].time;

minj=j;

}

}

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

pl[minj].pfn=INVALID;

pl[minj].time=-1;

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; //减少一个空闲页面

}

else

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

present\_time++;

}

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

}

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; //减少一个空闲页面

}

else

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

present\_time++;

}

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

}

void NUR( total\_pf) //NUR算法

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];

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;

}

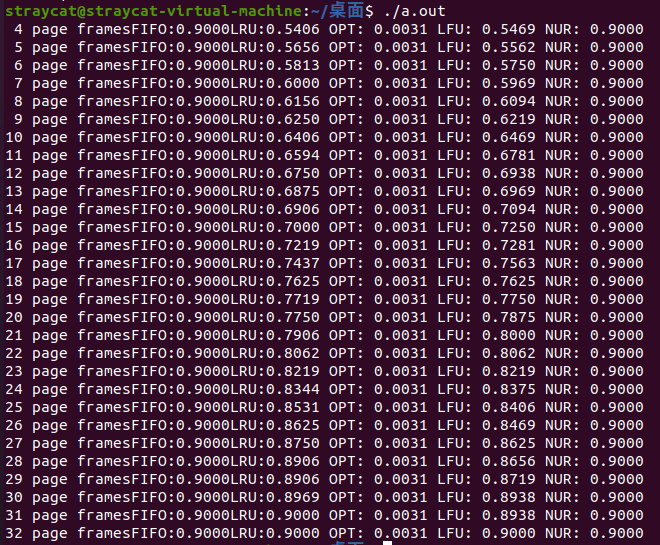
}

}

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

}

## 七、实验结果（附上运行结果的截图）



## 八、实验总结分析

这次的实验主要是用C语言来模拟实现页面调度算法，是比较难的一次挑战，虽然期间碰到了很多问题，但是经过多方咨询，查阅相关资料，最终解决了大部分问题，还是很有收获感和成就感。

其中代码部分，由于使用的是Ubuntu平台，与平时的Windows平台还是有一些差异，另外就是对工具的不熟悉，比如里面使用了C++的相关语句，但是使用了gcc来编译，结果报错，后来查阅相关资料才知道C++应当使用g++来编译。

在此同时也感谢这学期以来杨老师的认真教学与耐心指导，杨老师是一位很负责的老师，能够您的学生，我感到非常荣幸。

|  |  |
| --- | --- |
| **学生实验 心得** | 这次的课程设计，主要是针对算法的，第一个课程设计，关于操作系统的作业调度，学会之后，我能够更加深刻的理解到操作系统这种巧妙的作业调度算法，它们能够大幅提高CPU的利用率，使得计算机的运行速度更快。第二个课程实际，关于页面调度算法，这是对于内存管理方面的算法，它们能够提高对内存的利用率，使得CPU对数据的获取，能更加高效快速，对内存的管理优化，实际上也是对CPU的效率的一个更好的优化。  学生（签名）：彭重严  2020年12月30日 |
| **指导**  **教师**  **评语** | 成绩评定：  指导教师（签名）：  年 月 日 |

# 实验心得及教师评语