

大连理工大学

本科实验报告

课程名称： 操作系统实验

学院（系）： 电信学部

专 业： 计算机科学与技术

班 级： 电计 1604 班

学 号： 201624229

学生姓名： 盛君如

2019 年 06 月 13 日

实验项目列表

序号	实验项目名称	学时	成 绩			指导教师
			预习	操作	结果	
1	进程管理					
2	处理器调度					
3	存储管理					
4	磁盘移臂调度算法					
5	文件管理					
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
总计	学分：					

大连理工大学实验报告

学院（系）：____ 电信学部 ____ 专业：____ 计算机科学与技术 ____ 年级：____ 2016 届 ____

班级：____ 电计 1604 班 ____ 姓 名：____ 盛君如 ____ 学号：____ 201624229 ____

实验时间：____ 2019.05 ____ 实验室：____ 综一 401 ____ 指导教师：____ 杨志豪 ____

实验一、进程管理

1、实验目的

加深对于进程并发执行概念的理解。实践并发进程的创建和控制方法。观察和体验进程的动态特性。进一步理解进程生命期期间创建、变换、撤销状态变换的过程。掌握进程控制的方法，了解父子进程间的控制和协作关系。练习 Linux 系统中进程创建与控制有关的系统调用的编程和调试技术。

2、实验说明

进程可以通过系统调用 `fork()` 创建子进程并和其子进程并发执行。子进程初始的执行映像是父进程的一个复本。子进程可以通过 `exec()` 系统调用族装入一个新的执行程序。父进程可以使用 `wait()` 或 `waitpid()` 系统调用等待子进程的结束并负责收集和清理子进程的退出状态。

`fork()` 系统调用语法：

```
#include
```

```
pid_t fork(void);
```

`fork` 成功创建子进程后将返回子进程的进程号，不成功会返回-1。

`exec` 系统调用有一组 6 个函数，其中示例实验中引用了 `execve` 系统调用语法：

```
#include
```

```
int execve(const char *path, const char *argv[], const char *envp[]);
```

`path` 要装入的新的执行文件的绝对路径名字符串。`argv[]` 要传递给新执行程序的完整的命令参数列表(可以为空)。

`envp[]` 要传递给新执行程序的完整的环境变量参数列表(可以为空)。

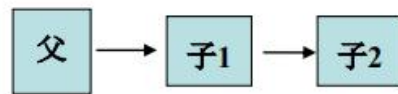
`Exec` 执行成功后将用一个新的程序代替原进程，但进程号不变，它绝不会再返回到调用进程了。如果 `exec` 调用失败，它会返回-1。

`getpid()` --- 获取进程的 `pid`

3、实验内容

3.1.1 内容一

- 每个进程都执行自己独立的程序，打印自己的 pid，每个父进程打印其子进程的 pid;



3.1.2 实验分析

本题的思想实现，首先是用 fork 函数创建父进程并且先执行，然后父进程创建一个子进程 p1，父进程根据自己的 pid>0 判断自己是父进程，打印出自己的 pid，然后其子进程 p1 根据 pid==0 判断自己是子进程，然后打印自己的 pid; 子进程 p1 接着又创建一个子子进程 p2，子进程作为新的父进程打印出自己的 pid, 子子进程 p2 依据 pid==0 判断自己是子子进程 p2 然后打印自己的 pid。

3.1.3 源程序

//实现实验的第一个内容：父进程->子进程 1->子进程 2

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    int pid,pid1;
    pid=fork();
    if(pid==0)//创建子进程
    {
        printf("其子进程 1 的 pid 为: %d.\n",getpid());
        pid1=fork();
        if(pid1==0)
        {
            printf("其子子进程 2 的 pid 为: %d.\n\n",getpid());
            printf("我是子子进程 2, 子子进程 2 的 pid 为: %d.\n",getpid());
        }
        else if(pid1>0)
        {
            printf("我是子进程 1, 子进程 1 的 pid 为: %d.\t",getpid());
        }
        else
        {
            //操作系统出错，可能进程数量达到上限或者其他原因
            printf("fork() error.\n");
            return -1;
        }
    }
    else if(pid>0)
```

```

        //创建父进程
    {
        printf("我是一个父进程,父进程的 pid 为: %d.\t",getpid());
    }
    else
    {
        //操作系统出错,可能进程数量达到上限或者其他原因
        printf("fork() error.\n");
        return -1;
    }
    return 0;
}

```

3.1.4 实验结果与截图

```

/cygdrive/g/test1
君如@DESKTOP-GRGGAKB ~
$ cd G:

君如@DESKTOP-GRGGAKB /cygdrive/g
$ cd test1

君如@DESKTOP-GRGGAKB /cygdrive/g/test1
$ ls
shell11  test1_1.c  test1_1.exe  test1_2.c  test1_2.exe

君如@DESKTOP-GRGGAKB /cygdrive/g/test1
$ ./test1_1
我是一个父进程,父进程的pid为: 11496.其子进程1的pid为: 4748.

君如@DESKTOP-GRGGAKB /cygdrive/g/test1
$ 我是子进程1,子进程1的pid为: 4748.其子子进程2的pid为: 9596.

我是子子进程2,子子进程2的pid为: 9596.

```

如截图所示,清晰地表现出父进程→子进程 1→子子进程 2 的关系,父进程的 pid: 11496,子进程 p1 的 pid: 4748,子子进程的 pid: 9596。内容显示与实验要求一致。

3.2.1 内容二

- 每个进程都执行自己独立的程序,打印自己的 pid,父进程打印其子进程的 pid;



3.2.2 实验分析

本实验实现的思路是父进程先后使用 fork 函数创建两个子进程,父进程通过 fork 返回的 pid 判断自己后,打印出子进程 p1 的 pid,子进程 p1 打印自身 pid。父进程通过 fork 创建子进程 p2,并打印出其 pid,子进程 p2 打印自身 pid。

3.2.3 源程序

```
//实现实验内容第二部分
#include<stdio.h>
#include<unistd.h>
int main()
{
    int pid1,pid2;
    /*这里创建了一个进程*/
    pid1=fork();
    if(pid1==0)
    {
        printf("我是第一个子进程 p1。子进程 p1 的 pid 为: %d.\n",getpid());
    }
    else if(pid1>0)
    {
        printf("我是父进程。我的第一个子进程 p1 的 pid 为: %d.\n",pid1);
        pid2=fork();
        if(pid2==0)
        {
            /*这里创建了一个子进程 2*/
            printf("我是第二个子进程 p2。子进程 p2 的 pid 为: %d.\n",getpid());
        }
        else if(pid2>0)
        {
            //
            printf("我是父进程。我的第二个子进程 p2 的 pid 为: %d.\n",pid2);
        }
        else
        {
            //操作系统出错，可能进程数量达到上限或者其他原因
            printf("fork() error.\n");
            printf("我是父进程,我的第二个子进程 p2 分配出错！");
        }
    }
    else
    {
        //操作系统出错，可能进程数量达到上限或者其他原因
        printf("fork() error.\n");
        printf("我是父进程,我的第一个子进程 p1 分配出错！");
        return -1;
    }
    return 0;
}
```

3.2.4 实验结果与截图

```

君如@DESKTOP-GRGGAKB /cygdrive/g/test1
$ ./test1_2
我是第一个子进程p1。子进程p1的pid为：2212.我是父进程。我的第一个子进程p1的pid为：2212.
我是父进程。我的第二个子进程p2的pid为：7400.我是第二个子进程p2。子进程p2的pid为：7400.

君如@DESKTOP-GRGGAKB /cygdrive/g/test1
$ ./test1_2
我是父进程。我的第一个子进程p1的pid为：7216.
我是第一个子进程p1。子进程p1的pid为：7216.
我是父进程。我的第二个子进程p2的pid为：10812.
我是第二个子进程p2。子进程p2的pid为：10812.

君如@DESKTOP-GRGGAKB /cygdrive/g/test1
$

```

如截图所示，清晰地表现出父进程→子进程 1、子进程 2 的关系，内容显示与实验要求一致。

3.3.1 内容三

编写一个命令处理程序，能处理 $\max(m, n)$, $\min(m, n)$ 和 $\text{average}(m, n, 1)$ 这几个命令(使用 `exec` 函数族)。

3.3.2 实验分析

在本次实验内容，我编写了一个命令处理程序 `shell` 模拟功能，同时配合 `max`、`min` 和 `average` 这三个函数，可以实现多次调用函数实现功能，我采用的是 `exec` 函数族中的 `execve` 函数实现系统中断调用功能。

我的 `shell` 程序有如下功能实现：

- 1、采用命令行参数，能够判断输入的命令、参数个数并调用正确的可执行文件；
- 2、如果输入为空，则一直显示 `JUNRU_CMD:`，继续等待用户的输入，直到输入有效；
- 3、对非法输入有判断功能，如命令不存在，参数个数不匹配等，进行提示
- 4、当调用可执行文件结束之后还要返回到 `shell` 中继续等待用户的输入，知道用户输入 `quit` 命令或者 `q` 命令时，显示 `See You!` 并退出。

`max`、`min`、`average` 三个函数我也做了相应的改进，以配合 `shell` 程序实现功能，让可执行文件执行结束之后回到 `shell`，我在 `max`、`min` 和 `ave` 函数结束之后又使用 `execve` 函数调用 `shell`，使之继续执行。

3.3.2 源程序

● `shell.cpp`)

```

#include <iostream>
#include <unistd.h>
#include <sstream>
#include <vector>
using namespace std;
int main(int argc, char *argv[])

```

```

{
    string cmd;
    cout<<"JUNRU_CMD:";
    while(getline(cin, cmd))
    {
        stringstream ss(cmd);
        vector<string> cmdlist;//定义 string 数组
        string temp;
        while(ss>>temp)//把 cmd 串压入数组中
        {
            cmdlist.push_back(temp);
        }
        if(cmdlist.empty())//cmd 为空时,
        {
            cout<<"JUNRU_CMD:";
            continue;
        }
        else if(cmdlist[0]=="max")//当为 max 时
        {
            if(cmdlist.size()>3)//判断非法性
            {
                cout<<"expected 2 arguments, but got
"<<cmdlist.size()-1<<endl;
                cout<<"JUNRU_CMD:";
                continue;
            }
            char arg1[50], arg2[50];
            strcpy(arg1, cmdlist[1].c_str());//cmdlist[1].c_str() 最后指向的
内容是垃圾, 因为 arg1 对象被析构, 所以不能直接利用 c_str 返回的字符串, 要利用
strcpy 等函数进行复制后再使用
            strcpy(arg2, cmdlist[2].c_str());
            char * argv[ ]={"./max", arg1, arg2, NULL};
            char * envp[ ]={NULL};
            execve("./max", argv, envp);////////////////////////////////////
        }
        else if(cmdlist[0]=="min")
        {
            if(cmdlist.size()>3)
            {
                cout<<"expected 2 arguments, but got
"<<cmdlist.size()-1<<endl;
                cout<<"JUNRU_CMD:";
                continue;
            }
            char arg1[50], arg2[50];
            strcpy(arg1, cmdlist[1].c_str());

```



```

        strcpy(arg2, cmdlist[2].c_str());
        char * argv[ ]={". /min", arg1, arg2, NULL};
        char * envp[ ]={NULL};
        execve(". /min", argv, envp);
    }
    else if(cmdlist[0]=="ave")
    {
        if(cmdlist.size()!=4)
        {
            cout<<"expected 3 arguments, but got
" <<cmdlist.size()-1<<endl;
            cout<<"JUNRU_CMD:";
            continue;
        }
        char arg1[50], arg2[50], arg3[50];
        strcpy(arg1, cmdlist[1].c_str());
        strcpy(arg2, cmdlist[2].c_str());
        strcpy(arg3, cmdlist[3].c_str());
        char * argv[ ]={". /ave", arg1, arg2, arg3, NULL};
        char * envp[ ]={NULL};
        execve(". /ave", argv, envp);
    }
    else if(cmdlist[0]=="quit" || cmdlist[0]=="q")
    {
        cout<<"See you!"<<endl;
        return 0;
    }
    else
    {
        cout<<"UNKNOW JUNRU_CMD:"<<cmdlist[0]<<endl;
    }
    cout<<"JUNRU_CMD:";
}
}

```

● max.c)

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    char * argvv[ ]={". /shell", NULL};
    char * envp[ ]={NULL};
    if(argc!=3)
        printf("expected 2 arguments, but got %d\n", argc-1);
    else

```

```

{
    int num1=atoi(argv[1]);
    int num2=atoi(argv[2]);
    if(num1>=num2)
        printf("max number is %d\n",num1);
    else
    {
        printf("max number is %d\n",num2);
    }
}
execve("./shell",argvv,envp);
return 0;
}

```

● min.c)

```

#include<stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    char * argvv[ ]={"./shell",NULL};
    char * envp[ ]={NULL};
    if(argc!=3)
    {
        printf("expected 2 arguments,but got %d\n",argc-1);
    }
    else
    {
        int num1=atoi(argv[1]);
        int num2=atoi(argv[2]);
        if(num1>=num2)
            printf("min number is %d\n",num2);
        else
        {
            printf("min number is %d\n",num1);
        }
    }
    execve("./shell",argvv,envp);
    return 0;
}

```

● ave.c)

```

#include <unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main(int argc, char *argv[])

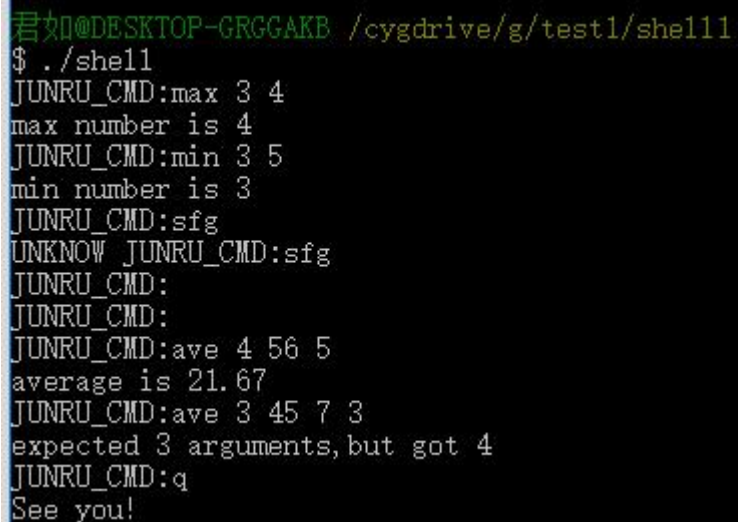
```

```

{
    char * argvv[ ]={". /shell",NULL};
    char * envp[ ]={NULL};
    if(argc!=4)
    {
        printf("expected 3 arguments,but got %d\n",argc-1);
    }
    else
    {
        float num1=atof(argv[1]);
        float num2=atof(argv[2]);
        float num3=atof(argv[3]);
        printf("average is %.2f\n", (num1+num2+num3)/3.0);
    }
    execve(". /shell", argvv, envp);
    return 0;
}

```

3.3.3 实验结果和分析



```

君如@DESKTOP-GRGGAKE /cygdrive/g/test1/shell11
$ ./shell
JUNRU_CMD:max 3 4
max number is 4
JUNRU_CMD:min 3 5
min number is 3
JUNRU_CMD:sfg
UNKNOWN JUNRU_CMD:sfg
JUNRU_CMD:
JUNRU_CMD:
JUNRU_CMD:ave 4 56 5
average is 21.67
JUNRU_CMD:ave 3 45 7 3
expected 3 arguments,but got 4
JUNRU_CMD:q
See you!

```

程序的测试结果如图所示，当输入 max、min 和 ave 等正确的命令都能正常识别并输出结果，当遇到空回车时 shell 自动忽略，当用户输入错误的命令时，例如 stg 错输入时，提示 unknown junru_cmd, 当用户输入的参数个数错误时，shell 进行提示参数个数与预期不匹配，输入 quit 或者 q 命令后 shell 程序退出。可见程序达到了预期的效果。

大连理工大学实验报告

学院（系）： 电信学部 专业： 计算机科学与技术 年级： 2016 届

班级： 电计 1604 班 姓 名： 盛君如 学号： 201624229

实验时间： 2019.05 实验室： 综一 401 指导教师： 杨志豪

实验二、处理器调度

一、 实验题目

随机给出一个进程调度实例，模拟进程调度，给出按照先来先服务算法 FCFS、轮转 RR、最短进程优先 SJF、最高响应比 HRN 进行调度各进程的完成时间、周转时间、带权周转时间。

二、 源程序

```
#include<stdio.h>
#include<iostream.h>
#include<stdlib.h>
#include<windows.h>
FILE *p_fcfs,*p_rr,*p_sjf,*p_hrn;
//*****
struct{
    char id;//进程名--
    int arrive_time;//到达时间--
    int serve_time;//服务时间--
    int remain_servetime;//剩余服务的时间
    int start_time;//开始执行时间
    int finish_time;//完成时间
    int route_time;//周转时间
    float rrt;//带权周转时间
}Process[20];//定义进程的相关结构信息
//*****

//下面是全局变量定义
//*****
int N;//总进程数目
int allover_time=0;//一共的执行时间
int ready_queue[20];//进程就绪队列
int time=0;//时间参数
int count_of_queue=0;//就绪队列中的进程数目
```

```

//*****

//初始化每个进程
void Initialize_Process()
{
    int i=0;
    for(;i<N;i++)
    {
        Process[i].start_time=0;
        Process[i].finish_time=0;
        Process[i].route_time=0;
        Process[i].rrt=0;
        Process[i].remain_servetime=Process[i].serve_time;
        allover_time+=Process[i].serve_time;//计算总共执行的时间
        ready_queue[i]=-1;//就绪队列为空
    }
}

//打印，将结果以美观的格式显示出来
void Print()
{
    float aver_rrt=0;//平均带权周转时间
    float aver_routetime=0;//平均周转时间
    printf("*****\n");
    printf("进程名  到达时间  服务时间  开始时间  完成时间  周转时间  带权周转\n");
    for(int i=0;i<N;i++)
    {
        aver_rrt+=Process[i].rrt;
        aver_routetime+=Process[i].route_time;
        printf("%2c\t",Process[i].id);
        printf("%5d\t",Process[i].arrive_time);
        printf("%5d\t",Process[i].serve_time);
        printf(" %5d\t",Process[i].start_time);
        printf(" %5d\t",Process[i].finish_time);
        printf("  %5d\t",Process[i].route_time);
        printf("    %7.2f\n",Process[i].rrt);
    }
    printf("\n\t 平均周转时间:%.2f\t 平均带权周转时\n",aver_routetime/N,aver_rrt/N);
    printf("*****\n");
}

```

```

//执行先来先服务算法
void fcfs()
{
    p_fcfs=fopen("./FCFS.txt","w+");
    printf("~~~~~\n");
    printf("          欢迎进入 FCFS 调度\n\n");
    printf("~~~~~\n");
    printf("          <时间序列>\n");
    printf("          ~~~~~\n");
    printf("-----\n");

    printf("时 间: ");
    for(int q=0;q<=30;q++)
    {
        if(q<9)
            printf("%d ",q);
        else
            printf("%d ",q);
    }
    printf("\n");
    printf("进程名: ");
    int earliest_arrive_time;//就绪队列中最早到达的进程的到达时间
    int pid;//模拟的最早到达的进程的进程号
    time=0;
    count_of_queue=0;
    while(time<=allover_time)
    {
        int i;
        count_of_queue=0;
        for(i=0;i<N;i++)//将到达时间在当前的 time 前的所有进程加入队列中
        {
            if(Process[i].arrive_time<=time &&
Process[i].remain_servetime!=0)
                //两个保证，一个是到达时间不能超过当前时间，一个是剩余服务时间不
能为 0
            {
                ready_queue[count_of_queue]=i;
                count_of_queue++;
            }
        }

        if(count_of_queue==0)//说明就绪队列为空
        {
            printf(" "); //记录过程

```

```

        time++;
    }
    else
    {
        //寻找此时最先到来的进程
        earliest_arrive_time=Process[ready_queue[0]].arrive_time;
        pid=ready_queue[0];
        for(i=1;i<count_of_queue;i++)
        {

if(earliest_arrive_time>Process[ready_queue[i]].arrive_time)//swap
        {

earliest_arrive_time=Process[ready_queue[i]].arrive_time;
        pid=ready_queue[i];
        }
        }
        Process[pid].start_time=time;//find the first process's start
time;

        int t1=time;
        time+=Process[pid].serve_time;
        for(;t1<time;t1++)
        {
            char ch=Process[pid].id;

            // fputc(ch,p_fcfs);/////
            fprintf(p_fcfs,"%c\n",ch);
            Sleep(100);
            printf("%c  ",Process[pid].id);//记录过程
        }
        Process[pid].finish_time=time;
        Process[pid].remain_servetime=0;

        Process[pid].route_time=Process[pid].finish_time-Process[pid].arrive_time
;

        Process[pid].rrt=(float)Process[pid].route_time/Process[pid].serve_time;
        }
    }
    printf("\n-----\n");
    -----\n");
    printf("\n");
    printf("          FCFS 先来先服务算法模拟\n");
    Print();//调用打印函数输出显示结果;
    fclose(p_fcfs);
}

```

```

void rr()
{
    p_rr=fopen("./RR.txt","w+");
    printf("~~~~~\n");
    printf("          欢迎进入 RR(q=1) 调度\n\n");
    printf("~~~~~\n");
    printf("          <时间序列>\n");
    printf("          ~~~~~\n");
    printf("-----\n");

    printf("时   间: ");
    for(int q=0;q<=30;q++)
    {
        if(q<9)
            printf("%d   ",q);
        else
            printf("%d ",q);
    }
    printf("\n");
    printf("进程名: ");
    count_of_queue=0;
    int flag=-1;//标记进程是否执行完毕, -1 为执行完毕, 否则为未执行完毕
    int i;
    time=0;
    while(time<=allover_time)
    {
        //将在当前 time 时刻到达的进程加入就绪队列
        for(i=0;i<N;i++)
        {
            if(Process[i].arrive_time==time)
            {
                ready_queue[count_of_queue++]=i;
            }
        }
        if(flag!=-1)//执行没有完毕, 把该编号加入队列尾部
        {
            ready_queue[count_of_queue++]=flag;
        }
        if(count_of_queue==0)//队列无进程
        {
            if(time<=allover_time)
                printf("   "); //记录过程
            time++;
        }
    }
}

```



```

    }
    else//取队首
    {

        if(Process[ready_queue[0]].remain_servetime==Process[ready_queue[0]].serve_time)

            Process[ready_queue[0]].start_time=time;//计算进程的开始时间

        if(Process[ready_queue[0]].remain_servetime<=1)//q=1, 进程将要结束, 两种情况, 一种是本次执行完后变成结束状态, 另一种是已经为结束状态
        {
            int t1=time;
            time+=Process[ready_queue[0]].remain_servetime;
            for(;t1<time;t1++)
            {
                char ch=Process[ready_queue[0]].id;
                fprintf(p_rr, "%c\n", ch);
                Sleep(100);
                printf("%c  ", Process[ready_queue[0]].id);//记录过程
            }
            Process[ready_queue[0]].finish_time=time;

            Process[ready_queue[0]].route_time=time-Process[ready_queue[0]].arrive_time;

            Process[ready_queue[0]].rrt=(float)Process[ready_queue[0]].route_time/Process[ready_queue[0]].serve_time;
            flag=-1;
        }
        else//进程未结束
        {
            time++;
            char ch=Process[ready_queue[0]].id;
            fprintf(p_rr, "%c\n", ch);
            Sleep(100);
            printf("%c  ", Process[ready_queue[0]].id);//记录过程
            Process[ready_queue[0]].remain_servetime--;
            flag=ready_queue[0];
        }
        //此时需要将进程调离出就绪队列
        for(i=0;i<(count_of_queue-1);i++)
        {
            ready_queue[i]=ready_queue[i+1];
        }
        count_of_queue--;
    }
}

```

```

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

printf("\n");
printf("RR 轮转(q=1)\n");
Print(); //调用打印函数输出显示结果;
fclose(p_rr);
}

void SJF() //算法思想和 FCFS 一样，只是参考的标准不一样
{
    p_sjf=fopen("./SJF.txt", "w+");
    printf("~~~~~\n");
    printf("欢迎进入 SJF 调度\n\n");
    printf("~~~~~\n");
    printf("    <时间序列>\n");
    printf("    ~~~~~\n");
    printf("-----\n");

    printf("时 间: ");
    for(int q=0;q<=30;q++)
    {
        if(q<9)
            printf("%d ", q);
        else
            printf("%d ", q);
    }
    printf("\n");
    printf("进程名: ");
    ///////////////////////////////////
    int shortest_serve_time; //就绪队列中服务时间最短的进程的服务时间
    int pid; //模拟的服务时间最短的进程的进程号
    time=0;
    count_of_queue=0;
    while(time<=allover_time)
    {
        int i;
        count_of_queue=0;
        for(i=0;i<N;i++) //将到达时间在当前的 time 前的所有进程加入队列中
        {
            if(Process[i].arrive_time<=time &&
Process[i].remain_servetime!=0)
                //两个保证，一个是到达时间不能超过当前时间，一个是剩余服务时间不
                能为 0

```

```

        {
            ready_queue[count_of_queue]=i;
            count_of_queue++;
        }
    }

    if(count_of_queue==0)//说明就绪队列为空
    {
        if(time<=allover_time)
            printf("    "); //记录过程
        time++;
    }
    else
    {
        //寻找此时最先到来的进程
        shortest_serve_time=Process[ready_queue[0]].serve_time;
        pid=ready_queue[0];
        //找出就绪队列中最短服务时间的进程
        for(i=1;i<count_of_queue;i++)
        {

            if(shortest_serve_time>Process[ready_queue[i]].serve_time)//swap
            {
                shortest_serve_time=Process[ready_queue[i]].serve_time;
                pid=ready_queue[i];
            }
        }
        Process[pid].start_time=time;//find the first process's start
time;

        int t1=time;
        time+=Process[pid].serve_time;
        for(;t1<time;t1++)
        {
            char ch=Process[pid].id;
            fprintf(p_sjf,"%c\n",ch);
            Sleep(100);
            printf("%c  ",Process[pid].id); //记录过程
        }
        Process[pid].finish_time=time;
        Process[pid].remain_servetime=0;

        Process[pid].route_time=Process[pid].finish_time-Process[pid].arrive_time
;

        Process[pid].rrt=(float)Process[pid].route_time/Process[pid].serve_time;
    }

```

```

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

    printf("\n");
    printf("          SJF 最短进程优先\n");
    Print(); //调用打印函数输出显示结果
    fclose(p_sjf);
}

//最高响应比优先算法
//思想:  $R=1+(\text{wait\_ime}/\text{serve\_time})$ , 其中 R 为优先权, 根据优先权来进行调度进程的
//执行
//记  $W\_S=\text{wait\_ime}/\text{serve\_time}$ 
void HRN()
{
    p_hrn=fopen("./HRN.txt", "w+");
    printf("-----\n");
    printf("          欢迎进入 HRN 调度\n\n");
    printf("-----\n");
    printf("          <时间序列>\n");
    printf("          ^^^^^^^^^\n");
    printf("-----\n");

    printf("时 间: ");
    for(int q=0;q<=30;q++)
    {
        if(q<9)
            printf("%d ", q);
        else
            printf("%d ", q);
    }
    printf("\n");
    printf("进程名: ");
    float W_S; ///记  $W\_S=\max(\text{wait\_ime}/\text{serve\_time})$ 
    int pid; //模拟的  $W\_S$  的进程的进程号
    time=0;
    count_of_queue=0;
    time=0;
    while(time<=allover_time)
    {
        int i;
        count_of_queue=0;
        for(i=0;i<N;i++) //将到达时间在当前的 time 前的所有进程加入队列中
        {

```

```

        if(Process[i].arrive_time<=time &&
Process[i].remain_servetime!=0)
        //两个保证，一个是到达时间不能超过当前时间，一个是剩余服务时间不
能为0
        {
            ready_queue[count_of_queue]=i;
            count_of_queue++;
        }
    }

    if(count_of_queue==0)//说明就绪队列为空
    {
        if(time<=allover_time)
            printf("    "); //记录过程
        time++;
    }
    else
    {
        W_S=(float)(time-Process[ready_queue[0]].arrive_time)/Process[ready_queue
[0]].serve_time;
        pid=ready_queue[0];
        ////找出就绪队列中 W_S 进程
        for(i=0;i<count_of_queue;i++)
        {

            if(W_S<(float)(time-Process[ready_queue[i]].arrive_time)/Process[ready_qu
eue[i]].serve_time)
            {

                W_S=(float)(time-Process[ready_queue[i]].arrive_time)/Process[ready_queue
[i]].serve_time;
                pid=ready_queue[i];
            }
        }
        Process[pid].start_time=time;//find the first process's start
time;
        int t1=time;
        time+=Process[pid].serve_time;
        for(;t1<time;t1++)
        {
            char ch=Process[pid].id;
            fprintf(p_hrn,"%c\n",ch);
            Sleep(100);
            printf("%c  ",Process[pid].id); //记录过程
        }
    }
}

```

```

        Process[pid].finish_time=time;
        Process[pid].remain_servetime=0;

Process[pid].route_time=Process[pid].finish_time-Process[pid].arrive_time
;

Process[pid].rrt=(float)Process[pid].route_time/Process[pid].serve_time;
    }
}
    printf("\n-----\n\n");
    printf("          HRN 最高响应比优先\n");
    Print();
    fclose(p_hrn);
}

void main()
{
    printf("本次运行的进程数目（不大于 20 个）：");
    cin>>N;
    cout<<"*****请输入有关进程
*****"<<endl;
    cout<<"请按照如右格式输入进程信息：<进程名> <到达时间> <服务时
间>"<<endl<<endl;
    for(int i=0;i<N;i++)
    {
        cout<<"第"<<i+1<<"个进程："<<' \t';
        cin>>Process[i].id>>Process[i].arrive_time>>Process[i].serve_time;
    }
    cout<<"*****"<<endl;
//进行 fcfs 算法
Initialize_Process();
fcfs();

//进行轮转法(q=1)
Initialize_Process();
rr();

//进行 SJF 最短进程优先
Initialize_Process();
SJF();

//HRN 最高响应比优先
Initialize_Process();
HRN();

```

```
}
```

Python 动画程序（以 FCFS 为例子）

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 11 19:33:17 2019
@author: 君如
"""

from tkinter import *
import numpy as np
import matplotlib.pyplot as plt
import time
FCFS=[]
btn=[]
btn0=[]
btnd=[]
with open(r"FCFS.txt","r") as f1:
    data1=f1.readlines()
for line in data1:
    FCFS.append(line.split()[0])

root = Tk()
root.title("电计 1604 班盛君如的<FCFS>")
a = LabelFrame(root, height=800, width=900, text='FCFS_Schedule')
a.pack(side='top', fill='both', expand=True)

sb = Scrollbar(root)    #垂直滚动条组件
sb.pack(side=RIGHT, fill=Y) #设置垂直滚动条显示的位置
listb=Listbox(root, yscrollcommand=sb.set)    #Listbox 组件添加 Scrollbar 组
件的 set() 方法

listb = Listbox(root)
for item in FCFS:                # 第一个小部件插入数据
    listb.insert(END, item)

len_fcfs=len(FCFS)
for i in range(0, len_fcfs):
    bttn=Button(a, text=i, heigh=3, width=5)
    bttn.grid(row=0, column=i)
    btn0.append(bttn)

for i in range(0, len_fcfs):
    btna=Button(a, text=i, heigh=3, width=5, fg='red')
    btna.grid(row=1, column=i)
    btnd.append(btna)
```

```

def Btn():
    for i in range(0, len_fcfs):
        if FCFS[i]=='A':
            btn1=Button(a, text=FCFS[i], heigh=3, width=5, bg='wheat')
        elif FCFS[i]=='B':
            btn1=Button(a, text=FCFS[i], heigh=3, width=5, bg='purple')
        elif FCFS[i]=='C':
            btn1=Button(a, text=FCFS[i], heigh=3, width=5, bg='blue')
        elif FCFS[i]=='D':
            btn1=Button(a, text=FCFS[i], heigh=3, width=5, bg='gray')
        elif FCFS[i]=='E':
            btn1=Button(a, text=FCFS[i], heigh=3, width=5, bg='cyan')
        btn1.grid(row=1, column=i)
        btn1.update()
        btn.append(btn1)
        time.sleep(0.5)

    btns=Button(a, text="FCFS", heigh=3, width=5, fg='red', bg='yellow', command=Bt
n)
    btns.grid(row=4, column=9)
    listb.pack(fill=BOTH)
    sb.config(command=listb.yview)
    root.mainloop()

```

三、 实验结果与截图

```

"C:\Users\君如\Desktop\操作系统实验(1)\2.处理器调度\Debug\处理器调度.exe"
本次运行的进程数目（不大于20个）： 5
*****请输入有关进程*****
请按照如右格式输入进程信息：<进程名> <到达时间> <服务时间>

第1个进程：    A 0 3
第2个进程：    B 2 6
第3个进程：    C 4 4
第4个进程：    D 6 5
第5个进程：    E 8 2
*****
~~~~~
                欢迎进入FCFS调度
~~~~~
                <时间序列>
~~~~~
时 间： 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
进程名： A  A  A  B  B  B  B  B  B  C  C  C  C  D  D  D  D  D  E  E
~~~~~

FCFS先来先服务算法模拟
*****
进程名  到达时间  服务时间  开始时间  完成时间  周转时间  带权周转时间
A        0         3         0         3         3         1.00
B        2         6         3         9         7         1.17
C        4         4         9        13         9         2.25
D        6         5        13        18        12         2.40
E        8         2        18        20        12         6.00

    平均周转时间:8.60      平均带权周转时间:2.56
*****

```



```

"C:\Users\君如\Desktop\操作系统实验(1)\2.处理器调度\Debug\处理器调度.exe"
欢迎进入RR(q=1)调度

<时间序列>

时间: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
进程名: A A B A B C B D C B E D C B E D C B D D

RR轮转(q=1)
*****
进程名 到达时间 服务时间 开始时间 完成时间 周转时间 带权周转时间
A      0      3      0      4      4      1.33
B      2      6      2      18     16     2.67
C      4      4      5      17     13     3.25
D      6      5      7      20     14     2.80
E      8      2     10     15      7     3.50

平均周转时间:10.80    平均带权周转时间:2.71
*****

```

```

"C:\Users\君如\Desktop\操作系统实验(1)\2.处理器调度\Debug\处理器调度.exe"
*****
欢迎进入SJF调度

<时间序列>

时间: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
进程名: A A A B B B B B B E E C C C C D D D D D

SJF最短进程优先
*****
进程名 到达时间 服务时间 开始时间 完成时间 周转时间 带权周转时间
A      0      3      0      3      3      1.00
B      2      6      3      9      7      1.17
C      4      4     11     15     11     2.75
D      6      5     15     20     14     2.80
E      8      2      9     11      3      1.50

平均周转时间:7.60    平均带权周转时间:1.84
*****

欢迎进入HRN调度

<时间序列>

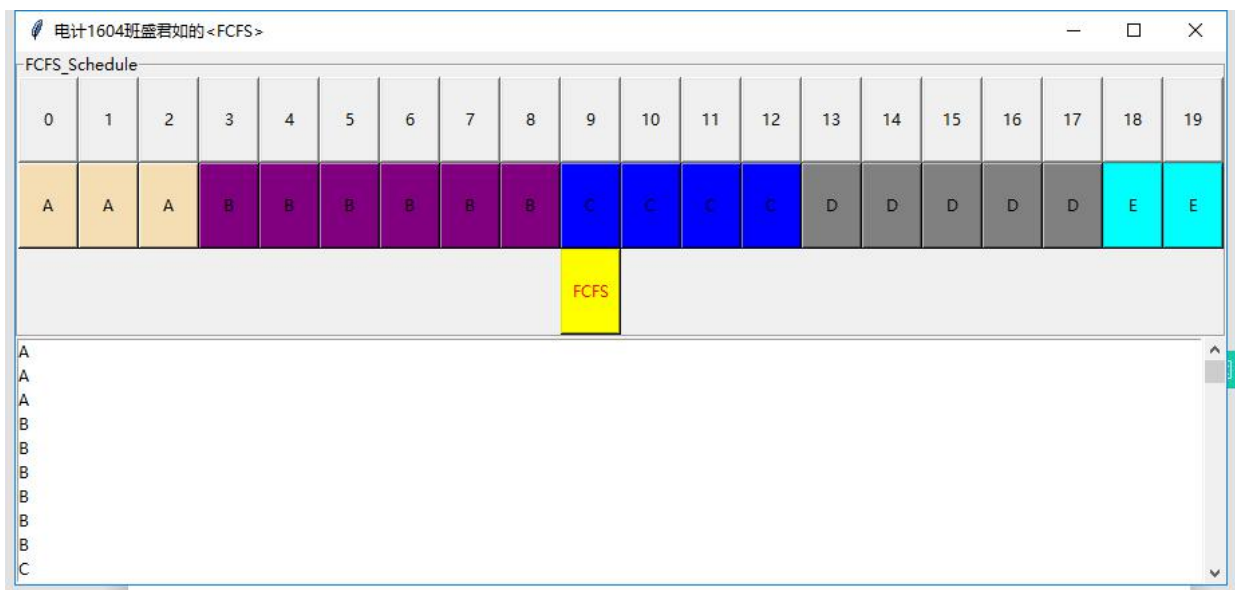
时间: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
进程名: A A A B B B B B C C C C E E D D D D D

HRN最高响应比优先
*****
进程名 到达时间 服务时间 开始时间 完成时间 周转时间 带权周转时间
A      0      3      0      3      3      1.00
B      2      6      3      9      7      1.17
C      4      4      9     13      9      2.25
D      6      5     15     20     14     2.80
E      8      2     13     15      7      3.50

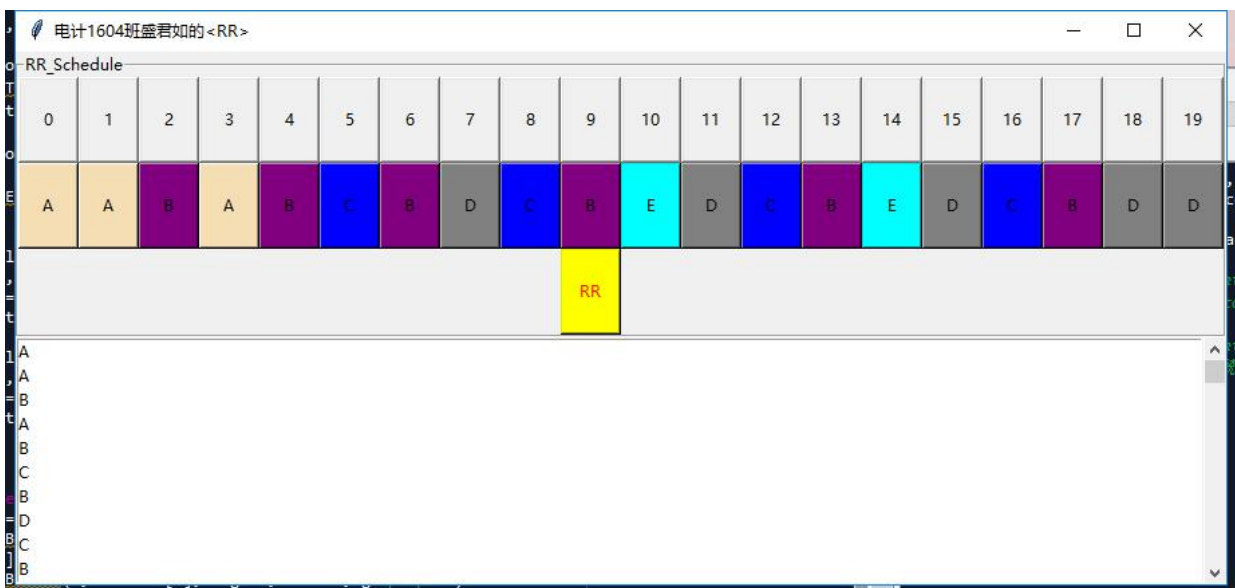
平均周转时间:8.00    平均带权周转时间:2.14
*****

```

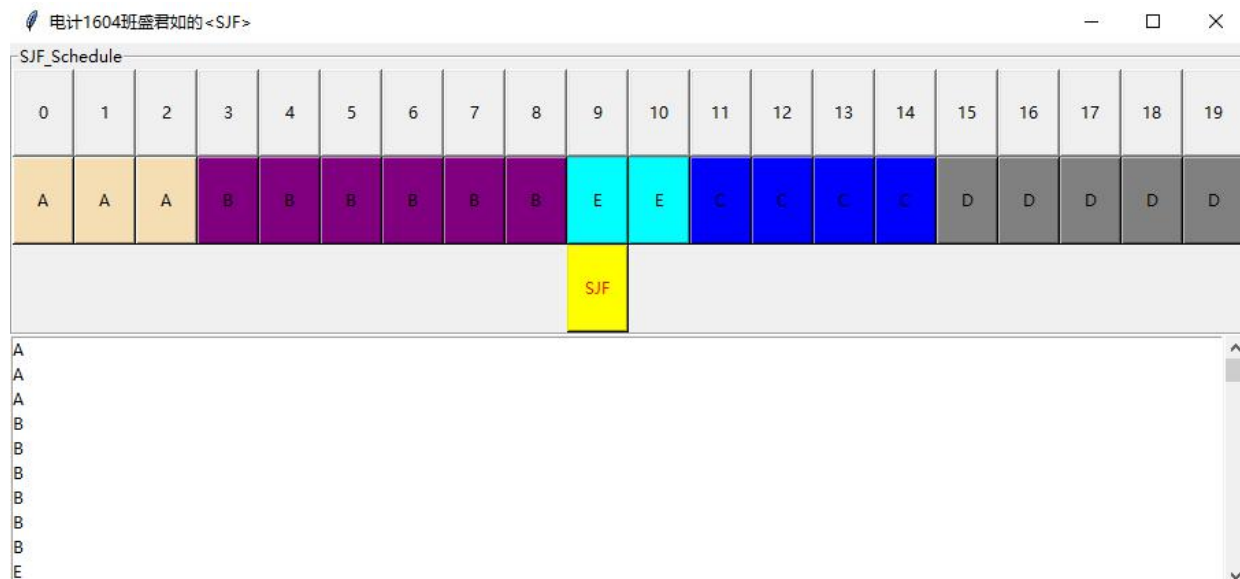
FCFS:



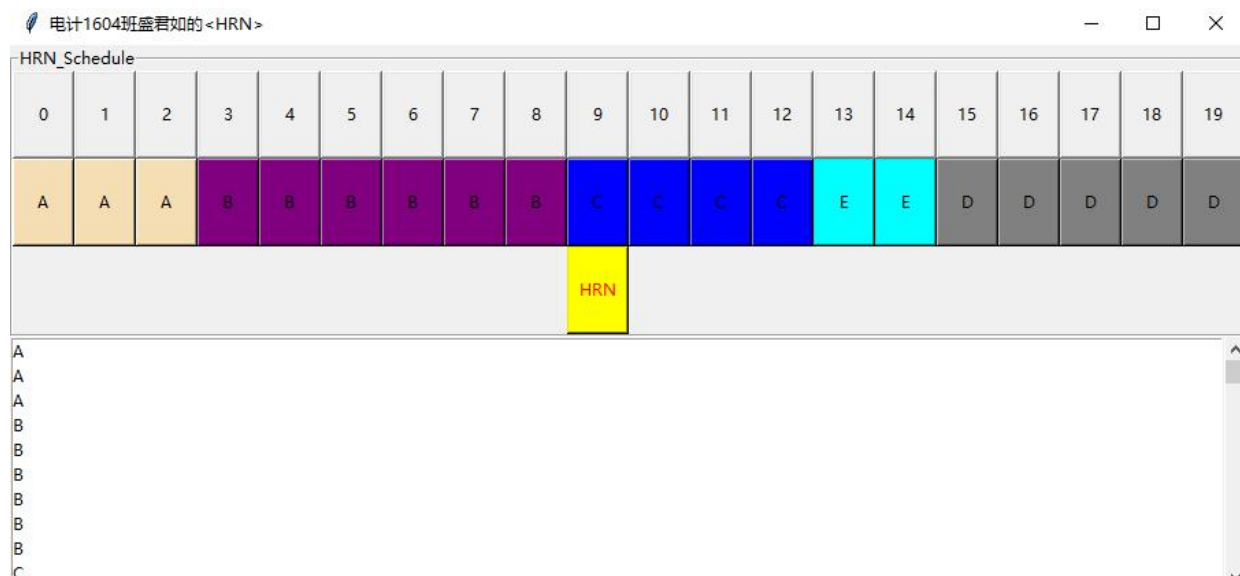
RR(q=1):



SJF:



HRN:



四、 实验感想

本次四个算法实现均采用了队列的思想，通过各自的思想标准来实现调度问题，如上图所示，很好地模拟了处理机调度的过程。也很好地锻炼了我的编程能力。

大连理工大学实验报告

学院（系）： 电信学部 专业： 计算机科学与技术 年级： 2016 届

班级： 电计 1604 班 姓 名： 盛君如 学号： 201624229

实验时间： 2019.05 实验室： 综一 401 指导教师： 杨志豪

实验三、存储管理

一、实验题目

模拟两种页面置换算法：LRU 算法和 FIFO 算法，给定任意的访问序列与页面数目，显示两种算法的页面置换过程，统计和报告不同置换算法依次淘汰的页号、缺页次数和缺页率。

二、源程序

```
#include<stdio.h>
#include<stdlib.h>
#include<windows.h>
#define N 50

int page_num;//内存中物理页面数目 m
int len;//页面引用序列长度
int ready_queue[N]; //存放页面引用序列的队列 a
int mem_queue[N]; //存放在主存中的序列的队列 b
int flag[N]; //缺页状态标记，0 为不缺页，1 为缺页
int miss_num;//缺页数 no
float miss_ratio;//缺页率 qyl
int f,r;//队列的首，尾

int checkup(int i)//检查内存中是否有这队列的页号，LRU 和 FIFO 共享使用
{
    for(int j=f;j<=r;j++)
    {
        if(mem_queue[j]==ready_queue[i])
            return 1;
    }
    return 0;
}

void Unfull_page(int i)//物理页面未满足时的页面调度，LRU 和 FIFO 共享使用
{

```

```

int j;
int k;
if(checkup(i)==0)//缺页时，将新页面加进去，不缺页时不用操作
{
    mem_queue[++r]=ready_queue[i];
    flag[i]=1;
}
printf("调入页号:%d",ready_queue[i]);
printf("\t");
for(j=f;j<=r;j++)//输出这次的过程
{
    Sleep(400);////////////////////////////////////////
    printf("%d ",mem_queue[j]);
}
k=page_num-(r-f+1);
switch(k)
{
    case 0:
    {
        printf(" \t 淘汰页:None\t 缺页: %d\n",flag[i]);
        break;
    }
    case 1:
    {
        printf(" \t 淘汰页:None\t 缺页: %d\n",flag[i]);
        break;
    }
    case 2:
    {
        printf(" \t 淘汰页:None\t 缺页: %d\n",flag[i]);
        break;
    }
}
}

```

```

void FIFO()
{
    int i,j;
    miss_num=0;
    miss_ratio=0.00;
    f=r=0;
    mem_queue[0]=ready_queue[0];
    flag[0]=1;
    printf("调入页号:%d",ready_queue[0]);//////////
    printf("\t%d",mem_queue[0]);
    for(i=0;i<page_num-2;i++)

```

```

{
    printf("\t");
}
printf("淘汰页:None\t 缺页: %d\n", flag[0]);
for(i=1; i<len; i++)
{
    flag[i]=0;
    //说明物理页面未满足
    if((r-f+1)<page_num)
    {
        Unfull_page(i);
    }
    else//说明物理页面已满
    {
        if(checkup(i)==0)//缺页
        {
            flag[i]=1;
            mem_queue[++r]=ready_queue[i];
            f++;
            printf("调入页号:%d", ready_queue[i]);
            printf("\t");
            for(j=f; j<=r; j++)//输出这次的过程
            {
                Sleep(400);////////////////////////////////////////
                printf("%d ", mem_queue[j]);
            }
            printf("\t 淘汰页:%d\t 缺
页: %d\n", mem_queue[f-1], flag[i]);
        }
        else//不缺页
        {
            printf("调入页号:%d", ready_queue[i]);
            printf("\t");
            for(j=f; j<=r; j++)//输出这次的过程
            {
                Sleep(400);////////////////////////////////////////
                printf("%d ", mem_queue[j]);
            }
            printf("\t 淘汰页:None\t 缺页: %d\n", flag[i]);
        }
    }
}
//计算缺页数
for(i=0; i<len; i++)
{

```

```

        if(flag[i]==1)
            miss_num++;
    }
    miss_ratio=(float)miss_num/len;
}

```

```

int search_LRU(int i)//查找最近使用的 m 个页面中最近最少使用的页号
{
    int j, t, cnt=0;
    int buff[N]; //数值为 1 表示最近有使用
    for(j=0; j<N; j++) //清 0
    {
        buff[j]=0;
    }
    for(t=i-1; t>=0; t--)
    {
        if(buff[ready_queue[t]]==0)
        {
            buff[ready_queue[t]]=1;
            cnt++;
            if(cnt==page_num) //达到了物理页面数目
            {
                for(j=f; j<=r; j++)
                {
                    if(mem_queue[j]==ready_queue[t])
                    {
                        return j;
                    }
                }
            }
        }
    }
    return 0;
}

```

//模拟 LRU 算法

```

void LRU()
{
    int i, j;
    miss_num=0;
    miss_ratio=0.00;
    f=r=0;
    mem_queue[0]=ready_queue[0];
    flag[0]=1;
    printf("调入页号:%d", ready_queue[0]);
    printf("\t%d", mem_queue[0]);
}

```

```

for(i=0;i<page_num-2;i++)
{
    printf("\t");
}
printf("淘汰页:None\t 缺页: %d\n", flag[0]);
//置换执行
for(i=1;i<len;i++)
{
    flag[i]=0;
    //说明物理页面未满，操作同 FIFO 一样
    if((r-f+1)<page_num)
    {
        Unfull_page(i);
    }
    else
    //说明物理页面已满，操作与 FIFO 不同
    {
        if(checkup(i)==0)//缺页
        {
            flag[i]=1;
            //替换
            int tmp=mem_queue[search_LRU(i)];//调用寻找 LRU 的索引
            mem_queue[search_LRU(i)]=ready_queue[i];
            printf("调入页号:%d", ready_queue[i]);
            printf("\t");
            for(j=f;j<=r;j++)//输出这次的过程
            {
                Sleep(400);////////////////////////////////////////
                printf("%d ", mem_queue[j]);
            }
            printf("\t 淘汰页:%d\t 缺页: %d\n", tmp, flag[i]);
        }
        else//不缺页
        {
            printf("调入页号:%d", ready_queue[i]);
            printf("\t");
            for(j=f;j<=r;j++)//输出这次的过程
            {
                Sleep(400);////////////////////////////////////////
                printf("%d ", mem_queue[j]);
            }
            printf("\t 淘汰页:None\t 缺页: %d\n", flag[i]);
        }
    }
}
}

```



```

//计算缺页数
for(i=0;i<len;i++)
{
    if(flag[i]==1)
        miss_num++;
}
miss_ratio=(float)miss_num/len;
}

void main()
{
    printf("          欢迎来到页面置换存储管理\n");
    printf("*****\n");
;
    printf("页面引用序列长度: ");
    scanf("%d",&len);
    printf("物理页面数目: ");
    scanf("%d",&page_num);
    printf("\n 请输入页面引用序列: \t");
    for(int i=0;i<len;i++)
    {
        scanf("%d",&ready_queue[i]);
    }
    printf("*****\n");
;
    printf("          *FIFO 算法*\n");
    printf("          *****\n\n");
    FIFO();
    printf("\n\t缺页次数: %d\t缺页率: %.2f\n",miss_num,miss_ratio);
    printf("*****\n");
;
    printf("          *LRU 算法*\n");
    printf("          *****\n\n");
    LRU();
    printf("\n\t缺页次数: %d\t缺页率: %.2f\n",miss_num,miss_ratio);
}

```

三、实验结果及分析

```
"C:\Users\君如\Desktop\操作系统实验(1)\3.存储管理\Debug\存储管理.exe"
欢迎来到页面置换存储管理
*****
页面引用序列长度: 10
物理页面数目: 3
请输入页面引用序列: 1 2 3 5 4 2 4 4 5 3
*****
                        *FIFO算法*
                        *****
调入页号:1           1           淘汰页:None       缺页: 1
调入页号:2           1 2         淘汰页:None       缺页: 1
调入页号:3           1 2 3       淘汰页:None       缺页: 1
调入页号:5           2 3 5       淘汰页:1           缺页: 1
调入页号:4           3 5 4       淘汰页:2           缺页: 1
调入页号:2           5 4 2       淘汰页:3           缺页: 1
调入页号:4           5 4 2       淘汰页:None       缺页: 0
调入页号:4           5 4 2       淘汰页:None       缺页: 0
调入页号:5           5 4 2       淘汰页:None       缺页: 0
调入页号:3           4 2 3       淘汰页:5           缺页: 1

                        缺页次数: 7      缺页率: 0.70
*****
                        *LRU算法*
                        *****
调入页号:1           1           淘汰页:None       缺页: 1
调入页号:2           1 2         淘汰页:None       缺页: 1
调入页号:3           1 2 3       淘汰页:None       缺页: 1
调入页号:5           5 2 3       淘汰页:1           缺页: 1
调入页号:4           5 4 3       淘汰页:2           缺页: 1
调入页号:2           5 4 2       淘汰页:3           缺页: 1
调入页号:4           5 4 2       淘汰页:None       缺页: 0
调入页号:4           5 4 2       淘汰页:None       缺页: 0
调入页号:5           5 4 2       淘汰页:None       缺页: 0
调入页号:3           5 4 3       淘汰页:2           缺页: 1

                        缺页次数: 7      缺页率: 0.70
Press any key to continue
```

给出十个序列，五个物理页面，同时得出如上结果对比，经过分析比较，符合实际情况。

四、讨论、建议、质疑

存储管理中的这两个调度算法，算法结构有很多相似性，但思想却不同，尤其是当页面满时，且缺页时的调度策略不同，导致如上不同的结果。

大连理工大学实验报告

学院（系）： 电信学部 专业： 计算机科学与技术 年级： 2016 届

班级： 电计 1604 班 姓 名： 盛君如 学号： 201624229

实验时间： 2019.05 实验室： 综一 401 指导教师： 杨志豪

实验四、磁盘移臂调度算法

一、实验题目

1. 示例实验程序中模拟两种磁盘移臂调度算法:SSTF 算法和 SCAN 算法;
2. 能对两种算法给定任意序列不同的磁盘请求序列, 显示响应磁盘请求的过程;
3. 能统计和报告不同算法情况下响应请求的顺序、移臂的总量

二、源程序

```
#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

#define N 30

int len;//磁盘访问序列长度

int begin;//读写头起始位置

int disk_queue[N];//磁盘访问序列的队列

int flag[N];//标记该磁道是否被访问, 1 表示被访问, 0 表示未被访问

int sum;//移臂总量


FILE* P_SSTF,* P_SCAN;

//////////

void SSTF()

{

// FILE *stream;

P_SSTF=fopen("SSTF.txt","w+");//C:\\Users\\君如\\Desktop\\操作系统实验\\4.磁盘调度算法
```

```

int i,tmp,min_distance;

int begin_SSTF=begin;

fprintf(P_SSTF,"%d\n",begin_SSTF);

sum=0;

for(i=0;i<len;i++)
{
    min_distance=INT_MAX;//定义最小距离并设定初始值

    for(int j=0;j<len;j++)//寻找最小距离
    {
        if(min_distance>abs(disk_queue[j]-begin_SSTF) && flag[j]==0)
        {
            min_distance=abs(disk_queue[j]-begin_SSTF);

            tmp=j;
        }
    }

    flag[tmp]=1;//tmp 为寻找的索引号

    sum+=min_distance;

    fprintf(P_SSTF,"%d\n",disk_queue[tmp]);

    printf("\t\t%d\t\t %d\n",disk_queue[tmp],min_distance);

    begin_SSTF=disk_queue[tmp];//调整磁道开始位置
}

fclose(P_SSTF);
}

//////////

void SCAN()
{
    P_SCAN=fopen("SCAN.txt","w+");//C:\Users\君如\Desktop\操作系统实验\4.磁盘调度算法

    int i,tmp,min_distance;

    int begin_SCAN=begin;

    fprintf(P_SCAN,"%d\n",begin_SCAN);

```

```

// int l_r;//判断方向，为 1 表示向右边进行，为 0 表示向左边方向进行

int len_l,len_r;//处在磁盘开始位置的左右边的磁道数目

sum=0;

len_r=len_l=0;

//计算处在磁盘开始位置的左右边的磁道数目

for(i=0;i<len;i++)
{
    if(begin_SCAN<=disk_queue[i])

        len_r++;

}

len_l=len-len_r;

for(i=0;i<len_r;i++)
{
    min_distance=INT_MAX;//定义最小距离并设定初始值

    for(int j=0;j<len;j++)//寻找最小距离

    {

        if((disk_queue[j]-begin_SCAN)>=0  &&  min_distance>(disk_queue[j]-begin_SCAN)

&& flag[j]==0)

        {

            min_distance=disk_queue[j]-begin_SCAN;

            tmp=j;

        }

    }

    flag[tmp]=1;

    sum+=min_distance;

    fprintf(P_SCAN,"%d\n",disk_queue[tmp]);//

    printf("\t\t%d\t\t  %d\n",disk_queue[tmp],min_distance);

    begin_SCAN=disk_queue[tmp];//调整磁道开始位置

}

for(i=0;i<len_l;i++)
{

```



```

        flag[i]=0;
    }

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

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

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

    printf("\t 访问的次磁道号\t 移动的磁道数\n");

    SSTF();

    printf("\n\t\tSSTF 移臂总量: %d\n",sum);

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

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

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

    printf("\t 访问的次磁道号\t 移动的磁道数\n");

    for(i=0;i<len;i++)
    {

        flag[i]=0;

    }

    SCAN();

    printf("\n\t\tSCAN 移臂总量: %d\n",sum);

}

```

Python 动画程序(以 STTF 为例):

```

# -*- coding: utf-8 -*-

"""

Spyder Editor

This is a temporary script file.

"""

import matplotlib.pyplot as plt,time

import numpy as np

# 绘制普通图像

sstf=[]

```

```

with open(r"SSTF.txt","r") as f1:

    data1=f1.readlines()

for line in data1:

    sstf.append(int(line.split()[0]))

for i in range(12):

    plt.ion()

    plt.clf()

    X=plt.plot(sstf[i+1:],list(range(1, len(sstf[i+1:]) + 1)),"b*-",linewidth=1) #在当前绘图对象绘图
(X 轴， Y 轴， 线类型， 线宽度)

    plt.xlabel("disk_sequence") #X 轴标签
    plt.ylabel("time") #Y 轴标签

    plt.title("SSTF_Schedule",fontsize='large',fontweight='bold',color='blue') #图标题

    plt.legend(X,'sstf')

    plt.pause(0.8)

    plt.ioff()

```

三、实验结果与分析

"C:\Users\君如\Desktop\操作系统实验(1)\4.磁盘调度算法\Debug\磁盘移臂调度算法.exe"

```

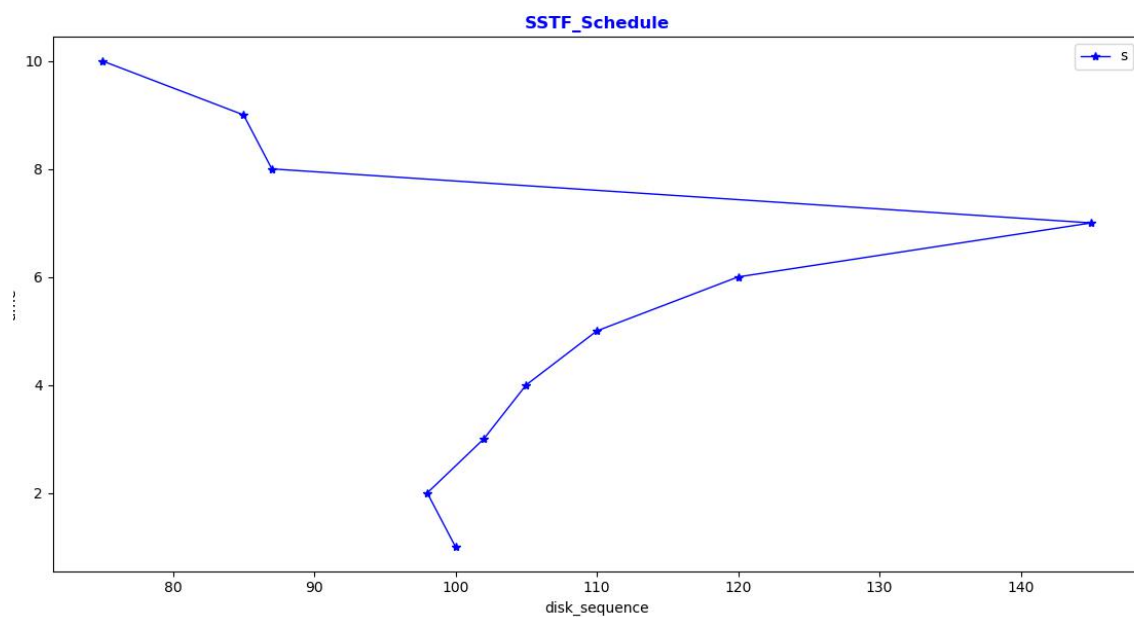
欢迎来到磁盘移臂调度
*****
磁盘访问序列长度: 9
读写头起始位置: 100
请输入磁盘访问序列: 105 98 87 120 145 110 102 85 75
*****
          *SSTF算法*
          *****

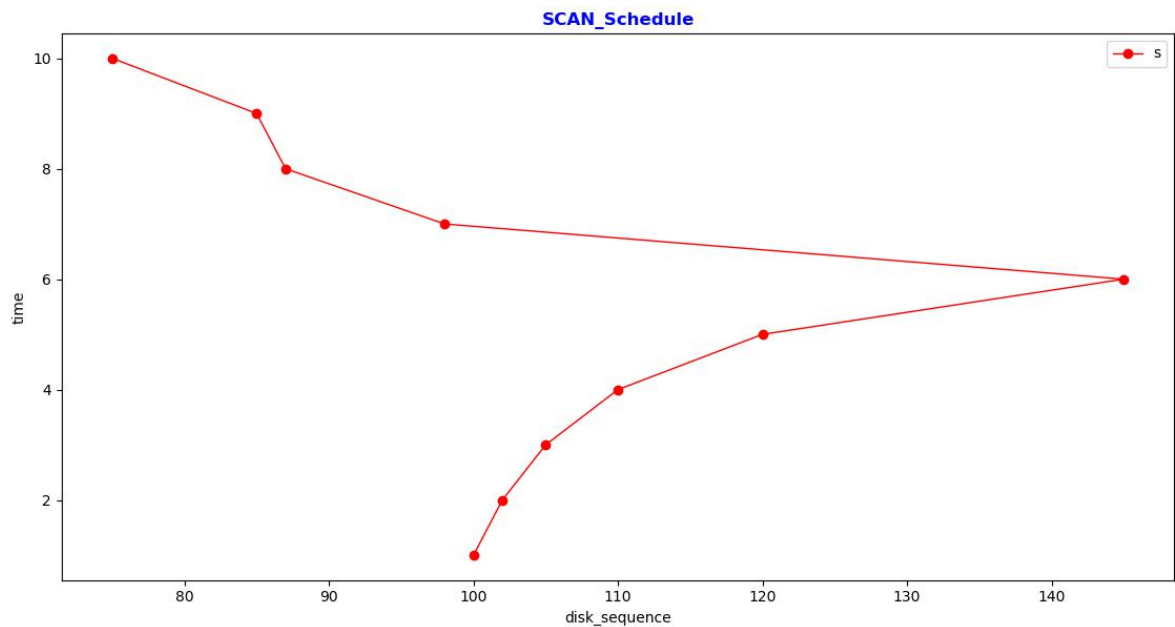
访问的次磁道号  移动的磁道数
          98      2
          102     4
          105     3
          110     5
          120    10
          145    25
          87     58
          85     2
          75     10

          SSTF移臂总量: 119
*****
          *SCAN算法*
          *****

访问的次磁道号  移动的磁道数
          102     2
          105     3
          110     5
          120    10
          145    25
          98     47
          87     11
          85     2
          75     10

          SCAN移臂总量: 115
Press any key to continue_
```





手动输入序列，得到如上图的次磁道号和移动磁道数以及移臂总数，并用 Python 实现动画效果，如上图清晰可见其工作过程。

四、讨论、建议、质疑

本次的磁盘调度算法较为简单，清晰可理解其工作原理以及工作过程，我也较好地模拟出了其工作过程，再接再厉。

大连理工大学实验报告

学院（系）： 电信学部 专业： 计算机科学与技术 年级： 2016 届

班级： 电计 1604 班 姓 名： 盛君如 学号： 201624229

实验时间： 2019.05 实验室： 综一 401 指导教师： 杨志豪

实验五、文件管理

一、实验题目

给出一个磁盘块序列:1、2、3、.....、500，初始状态所有块为 空的，每块的大小为 2k。使用空闲表、空闲盘区链、位示图 三种算法之一来管理空闲块。对于基于块的索引分配以下步骤:

(1) 随机生成 2k-10k 的文件 50 个，文件名为 1.txt、2.txt、.....、 50.txt，按照上述算法存模拟磁盘中。

(2) 删除奇数.txt(1.txt、3.txt、.....、49.txt)文件。

(3) 新创建 5 个文件(A.txt、B.txt、C.txt、DE.txt)，大小为:7k、5k、2k、9k、3.5k，按照与(1)相同的算法存储 到模拟磁盘中。

(4)给出文件 A.txt、B.txt、C.txt、D.txt、E.txt 的盘块存储状态和所有空闲区块的状态。

二、源程序

```
#include <iostream>

#include <sstream>

#include <vector>

#include <ctime>

#include <string>

using namespace std;

#define N1 50

#define N2 500

bool bitmap[N2]={0}; //位示图初始为 0，代表没有占位
```

```

int signal;//选择位示图
typedef struct filetable
{
    string filename;
    int indexed_block_node;//文件块索引号
}filetable;
vector<filetable> fat_Tab;//建立文件分配表数组

class Index_Block_Node
{
public:
    int Block[5];//文件的 5 个块的存放内容（其实是序号），可见运行结果
    Index_Block_Node()
    {
        for(int i=0;i<5;i++)
        {
            Block[i]=0;
        }
    }
    Index_Block_Node(int* x)
    {
        for (int i=0;i<5;i++)
        {
            Block[i]=*(x+i);
        }
    }
};

Index_Block_Node* M=new Index_Block_Node[N2];//定义 500 个磁盘块序列
void Createfiles()
{
    srand((unsigned)time(NULL));

```

```

for(int i=0;i<N1;i++)
{
    filetable file_buff;
    stringstream st;
    st<<(i+1);
    string str=st.str();
    file_buff.filename=str+".txt";
    for(int j=0;j<N2;j++)
    {
        if(bitmap[j]==false)//该块空闲
        {
            file_buff.indexed_block_node=j;
            bitmap[j]=true;
            //因为每块大小为 2K，所以我们随机产生 1—5 个块，并放入其中
            int blocks=(rand()%5)+1;
            for(int k=1;k<=blocks;k++)
            {
                M[file_buff.indexed_block_node].Block[k-1]=k+file_buff.indexed_block_node;
                //注意 《有修改，可能有错
                bitmap[k+file_buff.indexed_block_node]=true;
            }
            break;
        }
    }
    fat_Tab.push_back(file_buff);//把这次的新的分配表插入
}

}

void Delete_Odd_File()//删除奇数的文件占位
{

```

```

vector<filetable> ::iterator tmp=fat_Tab.begin();
for(int i=0;i<N1;i+=2)
{
    bitmap[tmp->indexed_block_node]=false;
    for(int j=0;j<5;j++)
    {
        int t=M[tmp->indexed_block_node].Block[j];
        if(t!=0)
        {
            bitmap[t]=false;
            M[tmp->indexed_block_node].Block[j]=0;
        }
        else
        {
            break;
        }
    }
    //删除对应的文件
    tmp=fat_Tab.erase(tmp);
    tmp++;
}
}

```

```

void New_File()
{
    string name[5]={"A. txt", "B. txt", "C. txt", "D. txt", "E. txt"};
    double size[5]={7, 5, 2, 9, 3.5};
    for(int i=0;i<5;i++)
    {
        filetable tmp;
        tmp.filename=name[i];
    }
}

```

```

for(int j=0;j<N2;j++)
{
    if(bitmap[j]==0)//表示未被占位
    {
        bitmap[j]=true;
        tmp.indexed_block_node=j;
        for(int k=0;k<size[i]/2.0;k++)
        {
            for(int t=j+1;t<N2;t++)
            {
                if(bitmap[t]==0)
                {
                    bitmap[t]=true;
                    M[tmp.indexed_block_node].Block[k]=t;
                    break;
                }
            }
        }
        break;
    }
}
fat_Tab.push_back(tmp);////把这次的新的分配表放入
}

```

```

void Print()
{
    vector<filetable> ::iterator tmp=fat_Tab.begin();
    cout<<"

```

```

-----\n";

    cout<<" .....<电计 1604 班
盛君如>.....\n";

    cout<<"

-----\n";

    cout<<' \t' <<' \t' <<' \t' <<" " <<"== 文件名==" <<"== 文件块索引号
==>" <<"===== 文件存放块号===== " <<endl;

    for(;tmp!=fat_Tab.end();tmp++)
    {
        cout<<' \t' <<' \t' <<' \t' <<' \t' <<" " <<tmp->filename<<' \t' <<' \t';
        printf("% 3d\t\t",tmp->indexed_block_node);
        for(int i=0;i<5;i++)
        {
            printf("%3d  ",M[tmp->indexed_block_node].Block[i]);
        }
        cout<<endl;
    }

    cout<<endl;

    cout<<"-----

-----\n";

    if(signal==1)

        printf(".....
... ..< 创 建 文 件 后 的 位 示
图>.....
\n");

    else

        if(signal==2)

```



```

printf(".....
...  ...  ...  ...  ...  <  删  除  文  件  后  的  位  示
图>.....
\n");

else

printf(".....
...  ...  ...  ...  ...  <  添  加  文  件  后  的  位  示
图>.....
\n");

cout<<"-----"
-----
-----\n";

cout<<"-----";

int u;
for(u=0;u<10;u++)
{
    cout<<"0"<<u<<" ";

}

for(;u<50;u++)
{
    cout<<u<<" ";

}

cout<<"-----"<<endl;

cout<<"-----"
-----
-----\n";

int j=0;

```

```

for(int i=0;i<N2-1;i++)
{
    if(i==0)
    {
        cout<<"--"<<"0"<<j<<"-- ";

    }
    j++;
    cout<<bitmap[i]<<" ";
    if((i+1)%50==0)
    {
        cout<<"-----";
        cout<<endl;
        cout<<"--0"<<j/50<<"-- ";
    }
}

cout<<bitmap[i]<<" ";
cout<<"-----";
cout<<endl;
cout<<"-----";
cout<<"-----"
-----

-----\n";

    cout<<endl;
}

int main()
{

    Createfiles();
    signal=1;

```

```
Print();  
Delete_Odd_File();  
signal=2;  
  
Print();  
New_File();  
signal=3;  
  
Print();  
return 0;  
}
```

三、实验结果与分析

```

=====
.....<电计1604班 盛君如>.....
=====
===文件名===文件块索引号=====文件存放块号=====
1.txt          0          1    0    0    0    0
2.txt          2          3    4    5    6    0
3.txt          7          8    9   10   11   12
4.txt         13         14   15   16   17    0
5.txt         18         19   20   21   22   23
6.txt         24         25   26    0    0    0
7.txt         27         28   29   30    0    0
8.txt         31         32    0    0    0    0
9.txt         33         34   35   36   37    0
10.txt        38         39   40   41    0    0
11.txt        42         43   44   45   46    0
12.txt        47         48   49   50   51   52
13.txt        53         54    0    0    0    0
14.txt        55         56   57   58    0    0
15.txt        59         60   61    0    0    0
16.txt        62         63    0    0    0    0
17.txt        64         65    0    0    0    0
18.txt        66         67   68   69   70    0
19.txt        71         72   73    0    0    0
20.txt        74         75   76   77    0    0
21.txt        78         79   80   81   82    0
22.txt        83         84   85    0    0    0
23.txt        86         87   88    0    0    0
24.txt        89         90   91    0    0    0
25.txt        92         93   94   95   96   97
26.txt        98         99    0    0    0    0
27.txt       100        101    0    0    0    0
28.txt       102        103  104  105  106  107
29.txt       108        109  110  111  112  113
30.txt       114        115    0    0    0    0
31.txt       116        117  118  119  120    0
32.txt       121        122  123  124  125  126
33.txt       127        128  129  130    0    0
34.txt       131        132    0    0    0    0
35.txt       133        134  135    0    0    0
36.txt       136        137    0    0    0    0
37.txt       138        139    0    0    0    0
38.txt       140        141  142    0    0    0
39.txt       143        144    0    0    0    0
40.txt       145        146    0    0    0    0
41.txt       147        148  149  150  151    0
42.txt       152        153  154  155  156  157
=====

```

[illegible]

```
"C:\Users\君如\Desktop\操作系统实验(1)\5.文件管理\Debug\文件管理.exe"
<电计1604班 盛君如>
=====
===文件名===文件块索引号=====文件存放块号=====
2.txt      2      3      4      5      6      7
4.txt      10     11     12     13     0      0
6.txt      19     20     21     22     0      0
8.txt      25     26     0      0      0      0
10.txt     33     34     35     0      0      0
12.txt     39     40     41     42     43     0
14.txt     49     50     51     52     53     0
16.txt     57     58     59     60     61     0
18.txt     64     65     66     67     68     0
20.txt     74     75     76     77     78     79
22.txt     85     86     87     0      0      0
24.txt     94     95     96     97     0      0
26.txt    100    101     0      0      0      0
28.txt    105    106    107     0      0      0
30.txt    113    114     0      0      0      0
32.txt    118    119    120    121     0      0
34.txt    124    125    126     0      0      0
36.txt    132    133    134    135    136     0
38.txt    143    144     0      0      0      0
40.txt    147    148    149    150     0      0
42.txt    157    158    159    160     0      0
44.txt    167    168    169     0      0      0
46.txt    173    174    175     0      0      0
48.txt    181    182    183     0      0      0
50.txt    188    189    190    191    192     0

<删除文件后的位示图>
-----00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49-----
--00-- 0 0 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1
--01-- 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0
--02-- 1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
--03-- 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0
--04-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--05-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--06-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--07-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--08-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--09-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

=====
===文件名===文件块索引号=====文件存放块号=====
2.txt      2      3      4      5      6      7
4.txt      10     11     12     13     0      0
6.txt      19     20     21     22     0      0
8.txt      25     26     0      0      0      0
10.txt     33     34     35     0      0      0
12.txt     39     40     41     42     43     0
14.txt     49     50     51     52     53     0
16.txt     57     58     59     60     61     0
18.txt     64     65     66     67     68     0
20.txt     74     75     76     77     78     79
22.txt     85     86     87     0      0      0
24.txt     94     95     96     97     0      0
26.txt    100    101     0      0      0      0
28.txt    105    106    107     0      0      0
30.txt    113    114     0      0      0      0
32.txt    118    119    120    121     0      0
34.txt    124    125    126     0      0      0
36.txt    132    133    134    135    136     0
38.txt    143    144     0      0      0      0
40.txt    147    148    149    150     0      0
42.txt    157    158    159    160     0      0
44.txt    167    168    169     0      0      0
46.txt    173    174    175     0      0      0
48.txt    181    182    183     0      0      0
50.txt    188    189    190    191    192     0
A.txt      0      1      8      9     14     0
B.txt      15     16     17     18     0      0
C.txt      23     24     0      0      0      0
D.txt      27     28     29     30     31     32
E.txt      36     37     38     0      0      0

<添加文件后的位示图>
-----00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49-----
--00-- 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1
--01-- 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0
--02-- 1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 0 0 0 1
--03-- 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0
--04-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--05-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--06-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--07-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--08-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

如上图所示，我采用了位示图的方法编写算法，实现了文件建立，删除，增加的功能。

四、讨论、建议、质疑

位示图法则是对于每个块使用 1bit 来记录这个块的当前使用状态,0 代表空闲,1 代表已经使用了,为了方便起见,本程序使用了一个长为 500 的布尔数组来记录每个块是否使用,增加文件时查看是否有连续且足够的位块,删除文件则较为简单,直接将相应的位的记录置 0。

实验感想

通过本次操作系统实验，从理论课走出来，进入实践环节，这是我们计算机专业动手编程能力的十分正确的教学模式，我也对操作系统中所学习的重要的算法有了更深层次的理解。

一开始我以为只是把算法实现了就可以了，但是后来发现其实并没有那么简单，算法的思想不理解深入，编程过程中很容易走偏，导致事倍功半。而且同学们的完成的作业非常认真且优秀，各种动画界面效果，很好地直观地展示出算法的运行过程，我也努力地去学习更好地完善自己的工作，在第一个实验中编写了 shell 程序使之能过多次调用 max、min、ave 函数。其他的实验也做了相应的动画效果。

台上三分钟，台下十年功，虽然在给老师演示时就短短几分钟，但是我私下十分努力去学习相应的程序编写，也得到了老师的认可，我也可以在自己的这门课程上画上完整的句号，这令我欣慰，计算机科学不同于其他学科，兼具理论和实践，而且是理论服务于实践，在老师的多次强调计算机学生要多锻炼自己的编程动手能力，我深有体会，感谢老师！