

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）： 电子信息与电气工程学部

专 业： 计算机科学与技术啊

班 级： 电计 1402

学 号： 201774009

学生姓名： 胡文博

2017 年 6 月 22 日

1.实验一：进程管理	1
1.1 题目一	1
1.1.1 题目分析	1
1.1.2 代码实现	1
1.1.3 实验结果	2
1.2 题目二	3
1.2.1 题目分析	3
1.2.2 代码实现	3
1.2.3 实验结果	4
1.3 题目三	4
1.3.1 题目分析	5
1.3.2 代码实现	5
1.3.3 实验结果	9
2.实验二：处理器调度	10
2.1 实验分析	11
2.2 代码实现	11
2.3 实验结果	18
3.实验三：存储管理上机作业	20
3.1 实验分析	21
3.2 代码实现	21
3.3 实验结果	25
4.实验四：磁盘移臂调度算法实验	28
4.1 实验分析	28
4.2 代码实现	28
4.3 实验结果	30
5.实验五：文件管理作业	31
5.1 实验分析	32
5.2 代码实现	32
5.3 实验结果	34
收获与体会	37

1. 实验一：进程管理

1.1 题目一

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；

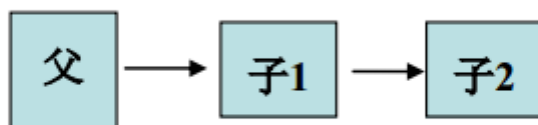


图 1.1

1.1.1 题目分析

题目的要求可分为三个：（1）父进程通过 fork 创建子进程，子进程再通过 fork 创建孙子进程（2）每个进程打印自己的 PID（3）父进程打印子进程 PID。所以进程可以通过 fork() 函数创建子进程，fork() 函数在父进程中的返回值是创建成功的子进程 PID 或者失败的负数返回值，fork() 函数在子进程中的返回值是 0，故可通过返回值判定该进程是子进程还是父进程。打印自己的 PID 可通过 getpid() 函数获得自身 PID 并打印。

1.1.2 代码实现

```
#include <iostream>
#include <unistd.h>
#include <cstdlib>
using namespace std;

int main(int argc, char const *argv[])
{
    pid_t pid;
    pid = fork(); // 创建子进程，并将返回值存入变量 pid
    if (pid < 0) // 创建失败
    {
        cout << "creat son process error!" << endl;
        exit(-1);
    }
    else if (pid == 0) // pid 为 0 代表该进程为子进程
    {
```

```
pid_t pid2 = fork();//子进程创建孙子进程
if (pid2 == 0)//pid2 为 0 代表该进程为孙子进程
{
    cout << "I am the grandson process, my ID is: " << getpid() << endl;
}
else if (pid2 < 0)//创建失败
{
    cout << "creat grandson process error!" << endl;
    exit(-1);
}
else//pid2 不为 0 代表该进程为孙子进程的父进程，即子进程
{
    cout << "I am the son process, my ID is: " << getpid() << ", and my son process (grandson process) ID is:" << pid2 << endl;
}
else//pid 不为 0 代表该进程为父进程
{
    cout << "I am the father process, my ID is: " << getpid() << ", and my son process ID is:" << pid << endl;;
}
return 0;
}
```

1.1.3 实验结果

实验结果如图 1.2 所示，可见代码实现了实验要求。

```
cris@cris-937: ~/gitRep/OS_Experiment/ProcessManange_ex1/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
cris@cris-937:~/gitRep/OS_Experiment/ProcessManange_ex1/build$ ./forkTest1
I am the father process, my ID is: 3221, and my son process ID is:3222
I am the son process, my ID is: 3222, and my son process (grandson process) ID is:3223
I am the grandson process, my ID is: 3223
cris@cris-937:~/gitRep/OS_Experiment/ProcessManange_ex1/build$
```

图 1.2

1.2 题目二

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;

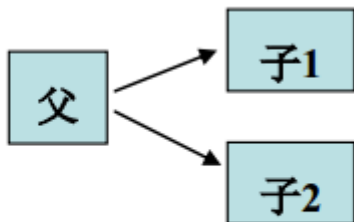


图 1.3

1.2.1 题目分析

此题与题目一的不同点在于父进程 `fork()` 创建了两个子进程,而不是子进程创建孙子进程。

1.2.2 代码实现

```
/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-05-15 18:37:23
 * @description: father -> son1
 *               -> son2
 */

#include <iostream>
#include <unistd.h>
#include <cstdlib>

using namespace std;
int main(int argc, char const *argv[])
{
    pid_t pid;
    pid = fork(); //创建子进程, 并将返回值存入变量 pid
    if (pid < 0) //创建失败
    {
        cout << "create son1 process error!" << endl;
        exit(-1);
    }
    else if (pid == 0) //pid 为 0 代表该进程为子进程
    {
        cout << "I am the son1 process, my ID is: " << getpid() << endl;
    }
}
```

```

else//pid 不为 0 代表该进程为父进程
{
    pid_t pid2 = fork();//父进程创建第二个子进程
    if (pid2 < 0)
    {
        cout << "creat son2 process error!" << endl;
        exit(-1);
    }
    else if (pid2 == 0)
    {
        cout << "I am the son2 process, my ID is: " << getpid() << endl;
    }
    else
    {
        cout << "I am the father process, my ID is: " << getpid() << ",
my first son ID is:"<<pid<<", and my second son process ID is:"<<pid2<<endl;
    }
}
return 0;
}

```

1.2.3 实验结果

实验结果如图 1.4 所示，可见代码实现了实验要求。

```

cris@cris-937: ~/gitRep/OS_Experiment/ProcessManange_ex1/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
cris@cris-937:~/gitRep/OS_Experiment/ProcessManange_ex1/build$ ./forkTest2
I am the father process, my ID is: 3287, my first son ID is:3288, and my second son process ID is:3289
I am the son1 process, my ID is: 3288
I am the son2 process, my ID is: 3289
cris@cris-937:~/gitRep/OS_Experiment/ProcessManange_ex1/build$ 

```

图 1.4

1.3 题目三

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)

1.3.1 题目分析

首先需要编写程序编译出 $\max(m, n)$, $\min(m, n)$, $\text{average}(m, n, l)$ 这三个可执行文件, 由于题目中未说明 m, n, l 的数据类型是整形还是浮点型, 故做如下设定, 若用户输入的数字含小数点则认为是浮点数, 结果做浮点运算, 否则按照整数运算。鉴于此前提, 这三个可执行文件的编写采用模板函数来完成, 代码见下节。

其次是命令处理程序, 即 shell 程序的编写, 我采用了正则表达式来进行词法解析, 解析出命令和参数, 然后根据命令使用 `exec1p()` 函数创建子进程执行相应命令。

1.3.2 代码实现

● $\max(m, n)$ 的实现:

```
/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-05-15 19:32:46
 * @description:
 */
#include <cstdlib>
#include <string>
#include <iostream>
#include <sstream>
using namespace std;

template <typename T>
inline T const& myMax (T const& a, T const& b) //模板函数实现任意数据类型的数据求最大值
{
    return a < b ? b : a;
}

bool isInt(string s) //根据输入数据是否小数点来判断是整形数还是浮点数
{
    stringstream sin(s);
    char c;
    while(sin>>c)
    {
        if(c == '.')
            return false;
    }
    return true;
}

int main(int argc, char *argv[])
{
    if (argc != 3) //参数个数不对
    {
        cout << "the number of parameters is error!" << endl;
        exit(-1);
    }
}
```

```

    string s1 = argv[1], s2 = argv[2];
    if(isInt(s1) && isInt(s2))//整形输入
    {
        cout<<"among "<<s1<<" and "<<s2<<" "<<"the bigger is:
"<<myMax(atoi(s1.data()),atoi(s2.data()))<<endl;
    }
    else//浮点型输入
    {
        cout<<"among "<<s1<<" and "<<s2<<" "<<"the bigger is:
"<<myMax(atof(s1.data()),atof(s2.data()))<<endl;
    }
    return 0;
}

```

● min(m,n)的实现:

```

/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-05-15 19:32:52
 * @description:
 */
#include <cstdlib>
#include <string>
#include <iostream>
#include <sstream>
using namespace std;

template <typename T>
inline T const& myMin (T const& a, T const& b)//模板函数实现任意数据类型的数据求最小值
{
    return a < b ? a : b;
}

bool isInt(string s)//根据输入数据是否小数点来判断是整形数还是浮点数
{
    stringstream sin(s);
    char c;
    while(sin>>c)
    {
        if(c == '.')
            return false;
    }
    return true;
}

int main(int argc, char *argv[])
{
    if (argc != 3)//参数个数不对
    {
        cout << "the number of parameters is error!" << endl;
        exit(-1);
    }
}

```



```

    string s1 = argv[1], s2 = argv[2];
    if(isInt(s1) && isInt(s2))//整形输入
    {
        cout<<"among "<<s1<<" and "<<s2<<", "<<"the smaller is:
"<<myMin(atoi(s1.data()),atoi(s2.data()))<<endl;
    }
    else//浮点型输入
    {
        cout<<"among "<<s1<<" and "<<s2<<", "<<"the smaller is:
"<<myMin(atof(s1.data()),atof(s2.data()))<<endl;
    }
    return 0;
}

```

● average(m, n, l)的实现:

```

/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-05-15 19:32:46
 * @description:
 */
#include <cstdlib>
#include <string>
#include <iostream>
#include <sstream>
using namespace std;

template <typename T>
inline T myAverage (T a, T b, T c)//模板函数实现任意数据类型的数据求平均值
{
    return (a + b + c)/3;
}

bool isInt(string s)//根据输入数据是否小数点来判断是整形数还是浮点数
{
    stringstream sin(s);
    char c;
    while (sin >> c)
    {
        if (c == '.')
            return false;
    }
    return true;
}

int main(int argc, char *argv[])
{
    if (argc != 4)//参数个数不对
    {
        cout << "the number of parameters is error, average func need 3
parameters!" << endl;
        exit(-1);
    }
}

```

```

string s1 = argv[1], s2 = argv[2], s3 = argv[3];
if (isInt(s1) && isInt(s2) && isInt(s3))//整形输入
{
    // cout<<"h"<<endl;
    cout << "the average of " << s1 << ", " << s2 << " and " << s3 << " is:
" << myAverage(atoi(s1.data()), atoi(s2.data()),atoi(s3.data())) << endl;
}
else//浮点型输入
{
    cout << "the average of " << s1 << ", " << s2 << " and " << s3 << " is:
" << myAverage(atof(s1.data()), atof(s2.data()),atof(s3.data())) << endl;
}
return 0;
}

```

● 命令处理程序 shell 的实现:

```

/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-05-15 19:52:46
 * @description:
 */
#include <iostream>
#include <unistd.h>
#include <cstdlib>
#include <string>
#include <vector>
#include <boost/algorithm/string.hpp>
#include <boost/format.hpp>
#include <boost/tokenizer.hpp>
using namespace std;
using namespace boost::algorithm;

int main(int argc, char const *argv[])
{
    cout << "welcome to young shell !" << endl;
    string s;
    cout << "young@shel: ->_->";
    vector<string> vecStr;
    while (getline(cin, s))//读入一行输入
    {
        cout << "young@shel: ->_->";//打印命令提示符
        vecStr.clear();
        boost::char_separator<char> sep(" , ()");
        typedef boost::tokenizer<boost::char_separator<char> >
        CustonTokenizer;
        CustonTokenizer tok(s, sep);//正则表达式将输入字符串分割为命令和参数
        for (CustonTokenizer::iterator beg = tok.begin(); beg != tok.end();
++beg)
        {
            vecStr.push_back(*beg);
        }
    }
}

```

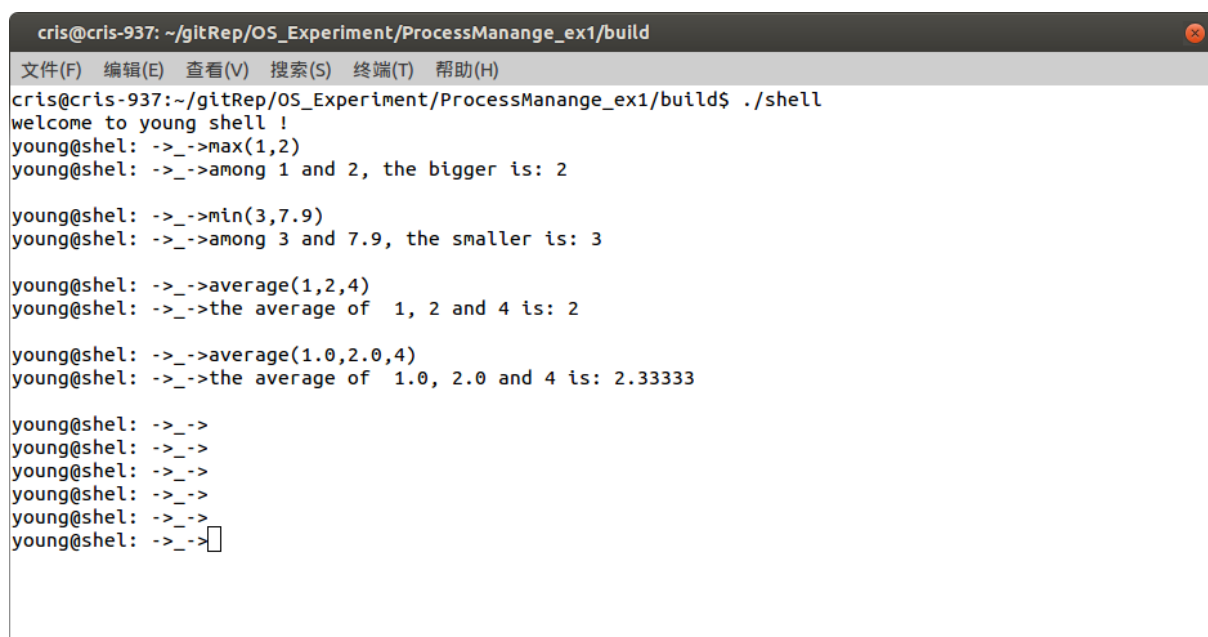
```

    if (vecStr.size()) //非空输入
    {
        if (fork() == 0) //判断是否位于子进程
        {
            if (vecStr[0] == "max") //子进程装入 max(m,n) 程序数据
            {
                execlp("/home/cris/gitRep/OS_Experiment/ProcessManange_ex1/build/max",
                    "./max", vecStr[1].data(), vecStr[2].data(), NULL);
                exit(0);
            }
            else if (vecStr[0] == "min") //子进程装入 min(m,n) 程序数据
            {
                execlp("/home/cris/gitRep/OS_Experiment/ProcessManange_ex1/build/min",
                    "./min", vecStr[1].data(), vecStr[2].data(), NULL);
                exit(0);
            }
            else if (vecStr[0] == "average") //子进程装入 average(m,n,l) 程序数据
            {
                execlp("/home/cris/gitRep/OS_Experiment/ProcessManange_ex1/build/average",
                    "./average", vecStr[1].data(), vecStr[2].data(), vecStr[3].data(), NULL);
                exit(0);
            }
            else
            {
                exit(0);
            }
        }
    }
    return 0;
}

```

1.3.3 实验结果

实验结果如图 1.5 所示，值得注意的是在 average() 函数中，输入的如果是 1, 2, 4 的话则按照整形数计算的均值是 2，而如果输入的是 1.0, 2.0, 4 则应该按照浮点数来计算，则结果应该是 2.33333，实验结果表明符合实验要求。



```
cris@cris-937: ~/gitRep/OS_Experiment/ProcessManange_ex1/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
cris@cris-937:~/gitRep/OS_Experiment/ProcessManange_ex1/build$ ./shell
welcome to young shell !
young@shel: ->_->max(1,2)
young@shel: ->_->among 1 and 2, the bigger is: 2

young@shel: ->_->min(3,7.9)
young@shel: ->_->among 3 and 7.9, the smaller is: 3

young@shel: ->_->average(1,2,4)
young@shel: ->_->the average of 1, 2 and 4 is: 2

young@shel: ->_->average(1.0,2.0,4)
young@shel: ->_->the average of 1.0, 2.0 and 4 is: 2.33333

young@shel: ->_->
young@shel: ->_->
young@shel: ->_->
young@shel: ->_->
young@shel: ->_->
young@shel: ->_->
```

图 1.5

2. 实验二：处理器调度

随机给出一个进程调度实例，如图 2.1 所示，模拟进程调度，给出按照算法先来先服务 FCFS、轮转 RR（ $q=1$ ）、最短进程优先 SJF、最高响应比优先 HRN 进行调度各进程的完成时间、周转时间、带权周转时间。

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

图 2.1

2.1 实验分析

本次实验采用 C++ 语言实现，采用了面向对象的编程思想，首先抽象出来两个类 process 和 processSchema，其中 process 类实现了对进程的描述，包含了进程的基本信息以及进程的各种动作，processSchema 类则实现了从文件读取题目中给定的进程调度实例，并将各进程存储在一个 vector 中，实现了记录所有进程的各种信息的功能。然后分别实现各个调度算法来调度 processSchema 中的进程。

在 FCFS 调度算法中，使用一个队列的数据结构来存储就绪进程队列，每到来一个新进程就将其放在队尾，每执行完一个进程进行一次调度，每次调度的内容是将队首的进程拿出并为其分配 CPU 进行执行，直至执行完毕。

在 RR 调度算法中，同样使用一个队列来存储就绪的进程队列，不同的是调度的时机，在 RR 调度算法中每当一个时间片用尽便会进行一次调度，调度过程是将当前正在执行的进程从队首拿出置于队尾，再给队首的进程分配 CPU 进行执行，并且每当新创建一个进程时便将该进程加入队尾等待调度，直至队列中所有进程执行完毕。

在 SJF 调度算法中，为了便于随机存取使用了 vector 来存储就绪的进程队列，调度的时机仍是当有进程执行完毕时才进行调度，每次调度时选取所要求的服务时间最短的进程进行执行，直至执行完毕所有进程。

在 HRN 调度算法中，使用 vector 存储就绪的进程，调度时机是进程执行完毕时，不同的是选择分配 CPU 的进程时依据的不再是静态的进程的服务时间，而是动态的等待时间和需要服务的时间之和再除以需要服务时间作为调度的优先权，直至执行完毕所有进程。

2.2 代码实现

● process.h:

```
/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-06-04 15:28:14
 * @description:
 */
#ifndef PROCESS_H
#define PROCESS_H
#include<string>
using namespace std;
class process
{
    int ID;//进程 ID 号
    int workTime = 0;//已经执行的时间
```

```

public:
    int serviceTime, //进程需要的服务时间
        comeTime, //进程到来的时间
        finishTime; //进程执行结束的时间点
    string name; //进程名
    process();
    process(int ID, int serviceTime, int comeTime, string name);
    void run(); //执行进程
    bool isFinished(); //查询进程是否执行完毕
    void disp(); //打印进程信息
    void dispResult(); //打印执行结果
    int getTurnaroundTime(); //计算周转时间
    double getWeightedTurnaroundTime(); //计算带权周转时间
};

#endif // PROCESS_H

```

● process.cpp:

```

/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-06-04 15:27:50
 * @description:
 */
#include "process.h"
#include <iostream>
using namespace std;
process::process()
{
    this->workTime = 0;
    this->name = "process";
}
process::process(int ID, int serviceTime, int comeTime, string name = "process")
{
    this->ID = ID;
    this->serviceTime = serviceTime;
    this->comeTime = comeTime;
    this->name = name;
    this->workTime = 0;
    this->finishTime = 0;
}
void process::disp() //打印进程信息
{
    cout << "" << this->name << " " << this->ID << " " << this->comeTime
    << " " << this->serviceTime << endl;
}
void process::dispResult()
{
    cout << "" << this->name << " " << this->ID << " " << this->finishTime
    << "\t\t\t\b\b\b\b" << this->getTurnaroundTime() << "\t\t\t" <<
    this->getWeightedTurnaroundTime() << endl;
}

```

```

}
void process::run() //执行进程
{
    workTime++;
}
bool process::isFinished() //查询进程是否执行完毕
{
    if (workTime >= serviceTime)
    {
        return true;
    }
    return false;
}
int process::getTurnaroundTime() //计算周转时间
{
    return finishTime-comeTime;
}
double process::getWeightedTurnaroundTime() //计算带权周转时间
{
    return (double)getTurnaroundTime()/serviceTime;
}

```

● processSchema.h:

```

/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-06-04 15:28:33
 * @description:
 */
#ifndef PROCESSSCHMA_H
#define PROCESSSCHMA_H
#include "process.h"
#include <vector>
#include <string>
using namespace std;

class processSchma
{
    void readFromFile(string fileName); //从文件读取进程调度实例
public:
    vector<process> processVec; //存储进程
    processSchma();
    processSchma(string fileName);
    void disp(); //打印所有进程信息
    void dispResult(); //打印所有进程结果信息
};
#endif // PROCESSSCHMA_H

```

● processSchema.cpp:

```

/**

```

```

*
* @author:    胡文博
* @email:    huwenbo@mail.dlut.edu.cn
* @dateTime:  2017-06-04 14:58:47
* @description:
*/
#include "processschma.h"
#include <fstream>
#include <iostream>
using namespace std;
processSchma::processSchma()
{}
processSchma::processSchma(string fileName)
{
    readFromFile(fileName);
}
void processSchma::readFromFile(string fileName) //从文件读取进程调度实例
{
    fstream f;
    f.open(fileName);
    string s;
    getline(f,s);
    int ID = 1;
    while(f>>s)
    {
        string name = s;
        f>>s;
        int comeTime = atoi(s.data());
        f>>s;
        int serviceTime = atoi(s.data());
        process indexP(ID++,serviceTime,comeTime,name);
        this->processVec.push_back(indexP);
    }
}
void processSchma::disp() //打印所有进程信息
{
    cout<<"-----"<<endl
        <<"Name  ID  createTime  serviceTime"<<endl
        <<"-----"<<endl;
    for(vector<process>::iterator iter = processVec.begin(); iter !=
processVec.end(); ++iter)
    {
        iter->disp();
    }
    cout<<"-----"<<endl;
}
void processSchma::dispResult() //打印所有进程结果信息
{
    cout<<"-----"<
<endl
        <<"Name  ID  finishTime  turnaroundTime  weightedTurnaroundTime"<<endl

```



```

<<"-----"<<endl;
    for(vector<process>::iterator iter = processVec.begin(); iter !=
processVec.end(); ++iter)
    {
        iter->dispResult();
    }

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

```

● main.cpp:

```

/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-06-03 21:18:17
 * @description:
 */
#include <iostream>
#include "process.h"
#include "processschma.h"
#include <queue>
#include <boost/circular_buffer.hpp>
#include <map>
using namespace std;
//检查是否有进程到来, 进程采用 queue 存储
void checkComeProcess( queue<process*> &comeonProcess, processSchma& schema,
int time )
{
    for (vector<process>::iterator iter = schema.processVec.begin(); iter !=
schema.processVec.end(); iter++)
    {
        if ( iter->comeTime == time )
        {
            comeonProcess.push(&(*iter));
        }
    }
}
//重载 checkComeProcess 函数, 区别是进程采用 vector 存储
void checkComeProcess( vector<process*> &comeonProcess, processSchma& schema,
int time )
{
    for (vector<process>::iterator iter = schema.processVec.begin(); iter !=
schema.processVec.end(); iter++)
    {
        if ( iter->comeTime == time )
        {
            comeonProcess.push_back( &(*iter));
        }
    }
}

```

```

}
//先来先服务算法的调度实现
void FCFS(processSchma& schema)
{
    int time = 0;
    queue<process*> comeonProcess;
    int processNum = schema.processVec.size();
    checkComeProcess(comeonProcess, schema, time);
    while (processNum) //进程还未执行完便继续
    {
        if (comeonProcess.empty()) //执行队列为空
        {
            time ++;
            checkComeProcess(comeonProcess, schema, time);
            continue;
        }
        comeonProcess.front()->run(); //执行队首程序
        time ++;
        checkComeProcess(comeonProcess, schema, time);
        if (comeonProcess.front()->isFinished()) //队首程序执行完毕
        {
            comeonProcess.front()->finishTime = time;
            comeonProcess.pop();
            processNum --;
        }
    }
}
//时间片轮转法调度算法实现，时间片为 1
void RR(processSchma& schema)
{
    int time = 0;
    int processNum = schema.processVec.size();
    queue<process*> comeonProcess;
    checkComeProcess(comeonProcess, schema, time);
    while (processNum) //进程还未执行完便继续
    {
        if (comeonProcess.empty()) //执行队列为空
        {
            time ++;
            checkComeProcess(comeonProcess, schema, time);
            continue;
        }
        auto tmp = comeonProcess.front(); //执行队首程序
        comeonProcess.pop();
        tmp->run();
        time++;
        checkComeProcess(comeonProcess, schema, time);
        if (tmp->isFinished()) //队首程序执行完毕
        {
            tmp->finishTime = time;
            processNum --;
        }
        else //将未执行完的队首程序置于队尾

```

```

        comeonProcess.push(tmp);
    }
}
//短进程优先调度算法实现
void SJF(processSchma& schema)
{
    int time = 0;
    vector<process*> comeonProcess;
    int processNum = schema.processVec.size();
    checkComeProcess(comeonProcess, schema, time);
    while (processNum) //进程还未执行完便继续
    {
        if (comeonProcess.empty()) //执行队列为空
        {
            checkComeProcess(comeonProcess, schema, ++time);
            continue;
        }
        auto tmp = comeonProcess.begin();
        //寻找服务时间最短的进程
        for (auto index = tmp; index != comeonProcess.end(); index++)
        {
            if ((*index)->serviceTime < (*tmp)->serviceTime)
                tmp = index;
        }
        auto tmpProcess = *tmp;
        comeonProcess.erase(tmp);
        while (!tmpProcess->isFinished() ) //直到该进程执行完毕
        {
            tmpProcess->run();
            checkComeProcess(comeonProcess, schema, ++time);
        }
        processNum--;
        tmpProcess->finishTime = time;
    }
}
//高响应比优先调度算法的实现
void HRN(processSchma& schema)
{
    int time = 0;
    vector<process*> comeonProcess;
    int processNum = schema.processVec.size();
    checkComeProcess(comeonProcess, schema, time);
    while (processNum) //进程还未执行完便继续
    {
        if (comeonProcess.empty()) //执行队列为空
        {
            checkComeProcess(comeonProcess, schema, ++time);
            continue;
        }
        auto tmp = comeonProcess.begin();
        // 寻找优先级最高的进程
        for (auto index = tmp; index != comeonProcess.end(); index++)
        {

```

```

        if ( ((time - (*index)->comeTime) / (double)(*index)->serviceTime) >
            ((time - (*tmp)->comeTime) / (double)(*tmp)->serviceTime) ) //compare
waittingtime/servicetime
            tmp = index;
    }
    auto tmpProcess = *tmp;
    comeonProcess.erase(tmp);
    while (!tmpProcess->isFinished() )//直到该进程执行完毕
    {
        tmpProcess->run();
        checkComeProcess(comeonProcess, schema, ++time);
    }
    processNum--;
    tmpProcess->finishTime = time;
}
}
// 主函数
int main(int argc, const char *argv[])
{
    if (argc != 3)//输入格式不对, 并提醒
    {
        cout << "please use the format: ProcesserSchema_ex2 [process list file]
[schema algorithm]" << endl;
        return -1;
    }
    processSchma pS(argv[1]);
    cout << "the given process list is:" << endl;
    pS.disp();
    map<string, function<void(processSchma&)>> > myEval;//建立字符串到函数的映射
    myEval["FCFS"] = FCFS;
    myEval["RR"] = RR;
    myEval["SJF"] = SJF;
    myEval["HRN"] = HRN;
    string comand(argv[2]);
    myEval[comand](pS);
    cout << "\n\n\nwork over, and the result is:" << endl;
    pS.dispResult();//打印结果
    return 0;
}

```

2.3 实验结果

实验结果如图所示, 其中图 2.2 是 FCFS 调度算法的结果, 图 2.3 是 RR 调度算法的结果, 图 2.4 是 SJF 调度算法的结果, 图 2.5 是 HRN 调度算法的结果。

```
cris@cris-937: ~/gitRep/OS_Experiment/ProcesserSchema_ex2/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
cris@cris-937:~/gitRep/OS_Experiment/ProcesserSchema_ex2/build$ ./ProcesserSchema_ex2 ../processList.txt FCFS
the given process list is:
-----
Name ID createTime serviceTime
-----
A    1    0         3
B    2    2         6
C    3    4         4
D    4    6         5
E    5    8         2
-----

work over, and the result is:
-----
Name ID finishTime turnaroundTime weightedTurnaroundTime
-----
A    1    3           3           1
B    2    9           7          1.16667
C    3   13           9          2.25
D    4   18          12          2.4
E    5   20          12           6
-----
cris@cris-937:~/gitRep/OS_Experiment/ProcesserSchema_ex2/build$
```

图 2.2

```
cris@cris-937: ~/gitRep/OS_Experiment/ProcesserSchema_ex2/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
cris@cris-937:~/gitRep/OS_Experiment/ProcesserSchema_ex2/build$ ./ProcesserSchema_ex2 ../processList.txt RR
the given process list is:
-----
Name ID createTime serviceTime
-----
A    1    0         3
B    2    2         6
C    3    4         4
D    4    6         5
E    5    8         2
-----

work over, and the result is:
-----
Name ID finishTime turnaroundTime weightedTurnaroundTime
-----
A    1    4           4          1.33333
B    2   18          16          2.66667
C    3   17          13          3.25
D    4   20          14          2.8
E    5   15           7          3.5
-----
cris@cris-937:~/gitRep/OS_Experiment/ProcesserSchema_ex2/build$
```

图 2.3

```

cris@cris-937: ~/gitRep/OS_Experiment/ProcesserSchema_ex2/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
cris@cris-937:~/gitRep/OS_Experiment/ProcesserSchema_ex2/build$ ./ProcesserSchema_ex2 ../processList.txt SJF
the given process list is:
-----
Name ID createTime serviceTime
-----
A    1    0         3
B    2    2         6
C    3    4         4
D    4    6         5
E    5    8         2
-----

work over, and the result is:
-----
Name ID finishTime turnaroundTime weightedTurnaroundTime
-----
A    1    3           3           1
B    2    9           7          1.16667
C    3   15          11          2.75
D    4   20          14          2.8
E    5   11           3          1.5
-----
cris@cris-937:~/gitRep/OS_Experiment/ProcesserSchema_ex2/build$ █

```

图 2.4

```

cris@cris-937: ~/gitRep/OS_Experiment/ProcesserSchema_ex2/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
cris@cris-937:~/gitRep/OS_Experiment/ProcesserSchema_ex2/build$ ./ProcesserSchema_ex2 ../processList.txt HRN
the given process list is:
-----
Name ID createTime serviceTime
-----
A    1    0         3
B    2    2         6
C    3    4         4
D    4    6         5
E    5    8         2
-----

work over, and the result is:
-----
Name ID finishTime turnaroundTime weightedTurnaroundTime
-----
A    1    3           3           1
B    2    9           7          1.16667
C    3   13           9          2.25
D    4   20          14          2.8
E    5   15           7          3.5
-----
cris@cris-937:~/gitRep/OS_Experiment/ProcesserSchema_ex2/build$

```

图 2.5

3. 实验三：存储管理上机作业

1. 示例实验程序中模拟两种置换算法： LRU 算法和 FIFO 算法。

2. 给定任意序列不同的页面引用序列和任意分配页面数目，显示两种算法的页置换过程。

3. 能统计和报告不同置换算法情况下依次淘汰的页号、缺页次数（页错误数）和缺页率。

3.1 实验分析

本次实验使用 C++ 实现，首先构建一个名为 memoryexpression 的类来模拟内存，该类实现了页表，以及记录系统为进程分配的物理块数。然后基于这个类来实现页面置换算法。

在 FIFO 算法中，使用一个大小为物理块数的队列来存放在内存中的页面，每当产生一次缺页中断时，首先判断是否还有空闲物理块来装入新的页面，若有则直接装入，否则选择队首的页面换出，并将新换入的页面至于队尾。

在 LRU 置换算法中，使用一个 vector 来存放在内存中的页面，每次访问页面时调整 vector 中的页面顺序，使最近最久未使用的页面至于 vector 的第一个位置，若产生缺页中断则首先判断是否还有空闲物理块来装入新的页面，若有则直接装入，否则选择 vector 第一个位置的页面换出，并将新换入的页面至于最后。

3.2 代码实现

● memoryexpression.h:

```
/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-06-05 21:24:09
 * @description:
 */
#ifndef MEMORYEXPRESSION_H
#define MEMORYEXPRESSION_H
#include <vector>
#include <queue>
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
class memoryExpression//内存表示类
{
public:
    vector<bool> pageList;//页表, bool 值表示该页是否在内存中
    vector<int> pageAccessOrder;//页面访问序列
    int blockNum = 0 ;//系统为进程分配的内存块数, 可由配置文件更改
    memoryExpression(string fileName);
```

```

    void dispPageAccessOrder(); //打印页面访问序列
    void disppageList(); //打印页表
};
#endif // MEMORYEXPRESSION_H

```

● memoryexpression.cpp:

```

/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-06-05 21:24:01
 * @description:
 */
#include "memoryexpression.h"
using namespace std;
memoryExpression::memoryExpression( string fileName )
{
    fstream f;
    f.open(fileName);
    string s;
    f>>s;
    f>>s;
    f>>s;
    f >> this->blockNum;
    f>>s,f>>s,f>>s;
    int tmp = 0;
    int maxPage = 0;
    while (f >> tmp)
    {
        this->pageAccessOrder.push_back(tmp);
        if(tmp>maxPage)
            maxPage = tmp;
    }
    for(int i = 0; i<maxPage; i++)
    {
        this->pageList.push_back(false);
    }
}

void memoryExpression::dispPageAccessOrder() //打印页面访问序列
{
    for (auto index = this->pageAccessOrder.begin(); index !=
pageAccessOrder.end(); index++)
    {
        cout << *index << endl;
    }
}

void memoryExpression::disppageList() //打印页表
{
    for (auto index = pageList.begin(); index != pageList.end(); index++)
    {
        cout << *index << endl;
    }
}

```


● main.cpp:

```

/**
 *
 * @author: 胡文博
 * @email: huwenbo@mail.dlut.edu.cn
 * @dateTime: 2017-06-05 18:21:20
 * @description:
 */
#include <iostream>
#include "memoryexpression.h"
#include <queue>
#include <map>
#include <functional>
using namespace std;
// 先进先出算法实现
void FIFO(memoryExpression& m)
{
    int missingPageNum = 0; // 缺页数
    queue<int> blocks;
    for (auto i = m.pageAccessOrder.begin(); i != m.pageAccessOrder.end(); i++)
    {
        cout <<
        "-----"
        << endl;
        cout << "access page " << *i << endl;
        if (m.pageList[*i]) // 不缺页
            continue;
        // 产生缺页中断
        missingPageNum++;
        if (blocks.size() < m.blockNum) // 有空闲内存块, 无需换出
        {
            blocks.push(*i);
            m.pageList[*i] = true;
            cout << "put page " << *i << " into memory block" << endl;
            continue;
        }
        m.pageList[blocks.front()] = false; // 没有空闲内存块, 需要换出
        m.pageList[*i] = true;
        cout << "move page " << blocks.front() << " out, and " << "put page "
        << *i << " into memory block" << endl;
        blocks.pop();
        blocks.push(*i);
    }
    cout << "\n\ntotal missing page number: " << missingPageNum << endl << "missing
page rate: " << ((double)missingPageNum / m.pageAccessOrder.size()) << endl;
}
// 最近最久未使用算法实现
void LRU(memoryExpression& m)
{
    int missingPageNum = 0;
    vector<int> blocks;
    for (auto i = m.pageAccessOrder.begin(); i != m.pageAccessOrder.end(); i++)

```

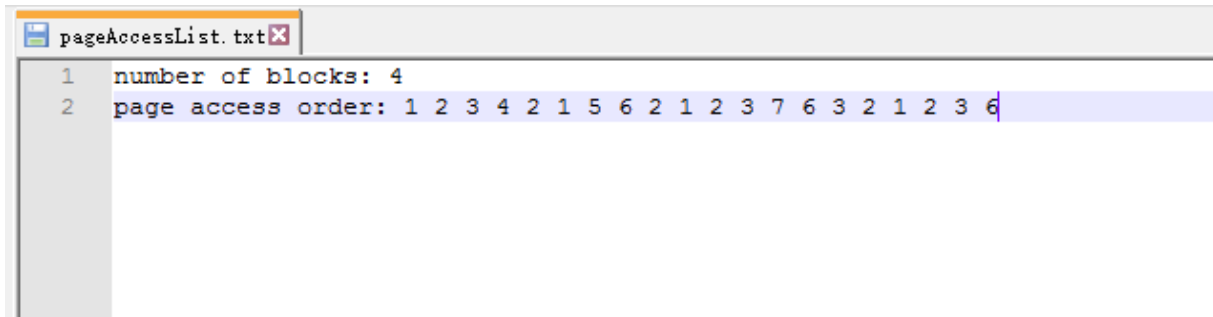
```

{
    cout <<
    "-----" << endl;
    cout << "access page " << *i << endl;
    if (m.pageList[*i]) //不缺页
    {
        for (auto it = blocks.begin(); it != blocks.end(); it++)
        {
            if (*it == *i)
            {
                blocks.erase(it);
                blocks.push_back(*i);
                break;
            }
        }
        continue;
    }
    // 产生缺页中断
    missingPageNum ++;
    if (blocks.size() < m.blockNum) //有空闲内存块, 无需换出
    {
        blocks.push_back(*i);
        m.pageList[*i] = true;
        cout << "put page " << *i << " into memory block" << endl;
        continue;
    }
    m.pageList[blocks.front()] = false; //没有空闲内存块, 需要换出
    m.pageList[*i] = true;
    cout << "move page " << blocks.front() << " out, and " << "put page "
    << *i << " into memory block" << endl;
    blocks.erase(blocks.begin());
    blocks.push_back(*i);
}
cout << "\n\ntotal missing page number: " << missingPageNum << endl << "missing
page rate: " << ((double)missingPageNum / m.pageAccessOrder.size()) << endl;
}
// 主函数
int main(int argc, char const *argv[])
{
    if (argc != 3)
    {
        cout << "please use the format: ProcesserSchema_ex2 [process list file]
[schema algorithm]" << endl;
        return -1;
    }
    memoryExpression m(argv[1]); //建立字符串到函数的映射
    map<string, function<void(memoryExpression& )> > myEval;
    myEval["FIFO"] = FIFO;
    myEval["LRU"] = LRU;
    string comand(argv[2]);
    myEval[comand](m);
    return 0;
}

```


3.3 实验结果

配置文件的页面访问序列如图 3.1 所示，图 3.2 是 FIFO 页面置换算法的结果，图 3.3 是 LRU 页面置换算法的结果。



```
1 number of blocks: 4
2 page access order: 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6
```

图 3.1



```
cris@cris-937: ~/gitRep/OS_Experiment/StorageManage_ex3/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
cris@cris-937:~/gitRep/OS_Experiment/StorageManage_ex3/build$ ./StorageManage_ex3 ../pageAccessList.txt FIFO
-----
access page 1
put page 1 into memory block
-----
access page 2
put page 2 into memory block
-----
access page 3
put page 3 into memory block
-----
access page 4
put page 4 into memory block
-----
access page 2
-----
access page 1
-----
access page 5
move page 1 out, and put page 5 into memory block
-----
access page 6
move page 2 out, and put page 6 into memory block
-----
access page 2
move page 3 out, and put page 2 into memory block
-----
access page 1
move page 4 out, and put page 1 into memory block
-----
access page 2
-----
access page 3
move page 5 out, and put page 3 into memory block
-----
access page 7
move page 6 out, and put page 7 into memory block
-----
access page 6
move page 2 out, and put page 6 into memory block
-----
access page 3
-----
access page 2
move page 1 out, and put page 2 into memory block
-----
access page 1
move page 3 out, and put page 1 into memory block
-----
access page 2
-----
access page 3
move page 7 out, and put page 3 into memory block
-----
access page 6

total missing page number: 14
missing page rate: 0.7
cris@cris-937:~/gitRep/OS_Experiment/StorageManage_ex3/build$ █
```

图 3.2



```
cris@cris-937: ~/gitRep/OS_Experiment/StorageManage_ex3/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
cris@cris-937:~/gitRep/OS_Experiment/StorageManage_ex3/build$ ./StorageManage_ex3 ../pageAccessList.txt LRU
-----
access page 1
put page 1 into memory block
-----
access page 2
put page 2 into memory block
-----
access page 3
put page 3 into memory block
-----
access page 4
put page 4 into memory block
-----
access page 2
-----
access page 1
-----
access page 5
move page 3 out, and put page 5 into memory block
-----
access page 6
move page 4 out, and put page 6 into memory block
-----
access page 2
-----
access page 1
-----
access page 2
-----
access page 3
move page 5 out, and put page 3 into memory block
-----
access page 7
move page 6 out, and put page 7 into memory block
-----
access page 6
move page 1 out, and put page 6 into memory block
-----
access page 3
-----
access page 2
-----
access page 1
move page 7 out, and put page 1 into memory block
-----
access page 2
-----
access page 3
-----
access page 6

total missing page number: 10
missing page rate: 0.5
cris@cris-937:~/gitRep/OS_Experiment/StorageManage_ex3/build$
```

图 3.3

4. 实验四：磁盘移臂调度算法实验

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。

4.1 实验分析

由于本次实验的结果在控制台不易表示，但是适合绘图表示，因此本次试验采用 Matlab 编写程序。

SSTF 调度算法中下一个访问点时距离当前磁臂位置最近的访问请求，可用 `min()` 函数寻找此最近点。

在 SCAN 调度算法中关键难点在于何时转换方向，分析后得出转换方向的条件为如果当前方向为向外且当前磁臂位置大于剩余访问序列中的最大值或者当前方向为向内且当前磁臂位置大于剩余访问序列中的最小值。

4.2 代码实现

● SSTF.m:

```
% /**
% *
% * @author:      胡文博
% * @email:       huwenbo@mail.dlut.edu.cn
% * @dateTime:    2017-06-12 16:01:06
% * @description: SSTF 调度算法
% */
function [responseVec,movingArmNum] = SSTF(startPoint,accessSequence)
    responseVec = zeros(length(accessSequence)+1);
    responseVec(1) = startPoint;%响应序列第一数为起始位置
    movingArmNum = 0;%记录移臂总数
    for i = 2:length(responseVec)
        %寻找距离当前磁臂最近的磁道访问请求
        [step,index] = min(abs(accessSequence-responseVec(i-1))) ;
        movingArmNum = movingArmNum + step;
        responseVec(i) = accessSequence(index);
        accessSequence(index) = [];
    end
end
```

● SCAN.m:

```
% /**
% *
% * @author:      胡文博
% * @email:       huwenbo@mail.dlut.edu.cn
```

```

% * @dateTime:      2017-06-12 16:00:49
% * @description:   SCAN 调度算法
% */
function [responseVec,movingArmNum] = SCAN(startPoint,accessSequence)
    Infinity = 1000000000000000; %正无穷
    responseVec = zeros(length(accessSequence)+1,1);
    responseVec(1) = startPoint; %响应序列第一数为起始位置
    movingArmNum = 0; %记录移臂总数
    % 确定首次移臂的方向
    [~,index] = min(abs(accessSequence-startPoint));
    if(accessSequence(index) >= startPoint)
        direction = 1;
    else
        direction = -1;
    end
    for i = 2:length(responseVec)
        % 如果当前方向为向外且当前磁臂位置大于剩余访问序列中的最大值
        % 或者当前方向为向内且当前磁臂位置大于剩余访问序列中的最小值
        % 则方向取反
        if( (direction==1 && responseVec(i-1) > max(accessSequence)) ...
            || (direction == -1 && responseVec(i-1) < min(accessSequence)) ...
        )
            direction = 0-direction;
        end
        tmp = accessSequence-responseVec(i-1);
        if(direction == 1)
            tmp(tmp<=0) = Infinity;
        else
            tmp(tmp>=0) = -Infinity;
        end
        %在不改变方向的前提下寻找距离当前磁臂最近的访问请求
        [step,index] = min(abs(tmp));
        movingArmNum = movingArmNum + step;
        responseVec(i) = accessSequence(index);
        accessSequence(index) = [];
    end
end
end

```

● draw_arrow.m:

```

% /**
% *
% * @author:      胡文博
% * @email:      huwenbo@mail.dlut.edu.cn
% * @dateTime:    2017-06-12 16:01:24
% * @description: 绘制带箭头的直线函数
% */
function out = draw_arrow(startpoint,endpoint,headsize,color)
    % accepts two [x y] coords and one double headsize
    v1 = headsize*(startpoint-endpoint)/2.5;
    theta = 22.5*pi/180;
    theta1 = -1*22.5*pi/180;
    rotMatrix = [cos(theta) -sin(theta) ; sin(theta) cos(theta)];

```

```

    rotMatrix1 = [cos(theta1) -sin(theta1) ; sin(theta1) cos(theta1)];
    v2 = v1*rotMatrix;
    v3 = v1*rotMatrix1;
    x1 = endpoint;
    x2 = x1 + v2;
    x3 = x1 + v3;
    hold on;
    % fill([x1(1) x2(1) x3(1)],[x1(2) x2(2) x3(2)],[0 0 0]);% this fills the
arrowhead (black)
    str=['(' num2str(x1(1)) ',' num2str(x1(2)) ')'];
    out = plot([startpoint(1) endpoint(1)],[startpoint(2)
endpoint(2)],'-o','linewidth',1,'color',color);
    set(gca,'ydir','reverse')
    text(x1(1)+0.2,x1(2)+0.1,str);

```

● main.m:

```

% /**
% *
% * @author:      胡文博
% * @email:      huwenbo@mail.dlut.edu.cn
% * @dateTime:    2017-06-12 16:00:26
% * @description:
% */
accessSequence = load ('diskAccessSequence.txt');%从配置文件导入磁道访问序列
startPoint = 53;%默认起始位置
% SCAN 调度并绘图
[responseVec,movingArmNum] = SCAN(startPoint,accessSequence);
for i = 2:length(responseVec)
    p1 = draw_arrow([responseVec(i-1),i-2 ], [responseVec(i),i-1],0.1,'g');
end
title(['total number of moving arm:',num2str(movingArmNum)])
% SSTF 调度并绘图
[responseVec,movingArmNum2] = SSTF(startPoint,accessSequence);
hold on;
for i = 2:length(responseVec)
    p2 = draw_arrow([responseVec(i-1),i-2 ], [responseVec(i),i-1],0.1,'b');
end
title(['total number of moving arm:']; ['SCAN ',num2str(movingArmNum),' ',
SSTF ',num2str(movingArmNum2)]])
legend([p1,p2], 'SCAN', 'SSTF')

```

4.3 实验结果

配置文件规定的磁盘访问序列如图 4.1 所示, 实验结果如图 4.2 所示, 可知完成了实验要求。

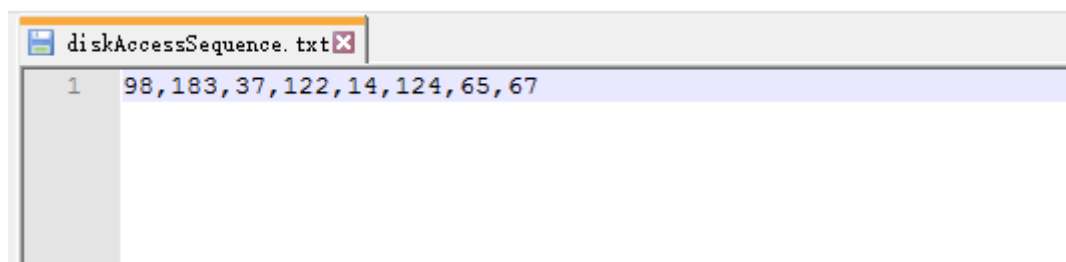


图 4.1

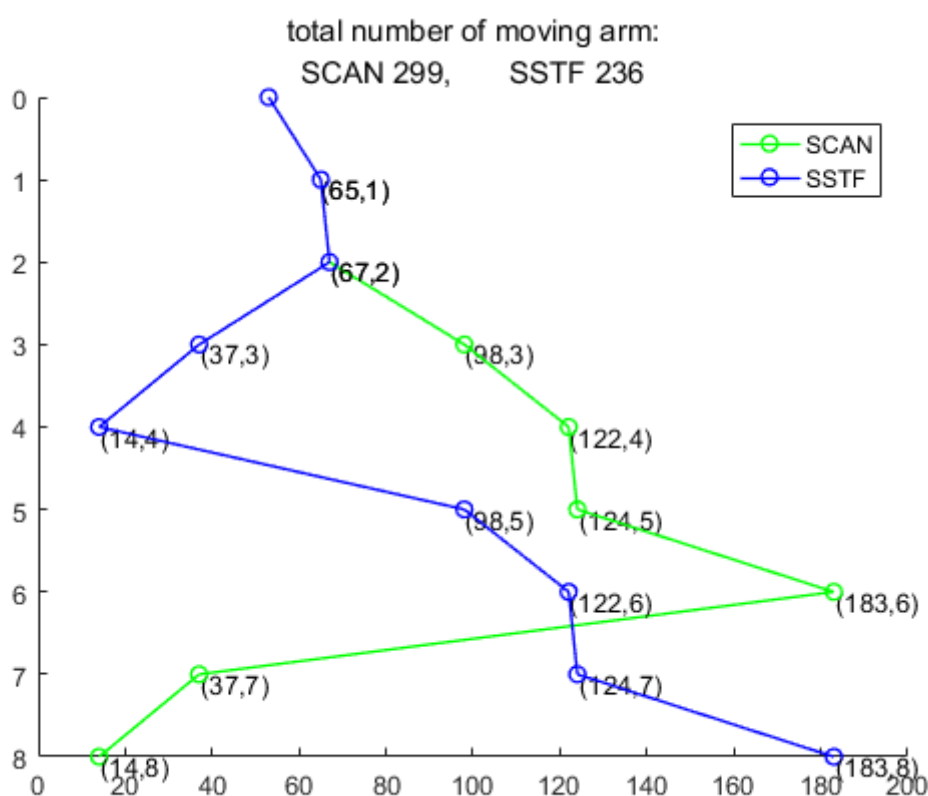


图 4.2

5. 实验五：文件管理作业

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。选择使用空闲表、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执行以下步骤：

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

（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、 D.txt、 E.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。

5.1 实验分析

本次实验采用位示图来管理空闲块，本实验的算法部分不难，难点在于可视化，故本实验采用 Matlab 来将结果更好地可视化。

分析实验可知，本实验的基础是实现存储文件的函数和删除文件的函数，而位示图则采用一个 25*20 的矩阵来表示。其中存储函数的实现需要寻找到第一个可以放下改文件的空闲区块，而删除文件的函数只需将该文件对应的位置 0 即可。

5.2 代码实现

● saveFile.m:

```
% /**
% *
% * @author:      胡文博
% * @email:       huwenbo@mail.dlut.edu.cn
% * @dateTime:    2017-06-11 22:58:16
% * @description: 存储文件函数
% */
function [bitMap,files] =saveFile( bitMapRaw, fileName, fileSize )
    bitMap = bitMapRaw;
    files.len = fileSize;
    files.name = fileName;
    fileSize = ceil(fileSize/2);
    cnt = 0;
    result = 0;
    %寻找第一块能够放下改文件的物理块序列
    for indexi = 1:size(bitMapRaw,1)
        for indexj = 1:size(bitMapRaw,2)
            if(bitMapRaw(indexi,indexj) > 0)
                cnt = 0;
                continue;
            end
            cnt = cnt+1;
            if(cnt >= fileSize)
                result = 1;
                break;
            end
        end
        if(result == 1)
            break;
        end
    end
end
```

```

if(result == 0)
    error(['Disk has no space for file',(fileName)]);
end
bitNum = size(bitMapRaw,2)*(indexi - 1)+indexj - (fileSize-1);
files.start = bitNum;
startj = mod(bitNum-1, size(bitMapRaw,2) ) + 1;
starti = (bitNum - startj)/size(bitMapRaw,2) + 1;
% 将该物理块序列的位示图置 1
for i = starti:indexi
    if (i == starti)
        s = startj;
    else
        s = 1;
    end
    if(i == indexi)
        e = indexj;
    else
        e = size(bitMap,2);
    end
    bitMap(i,s:e) = 1;
end
end
end

```

● deleteFile.m:

```

% /**
% *
% * @author:      胡文博
% * @email:      huwenbo@mail.dlut.edu.cn
% * @dateTime:    2017-06-11 22:57:40
% * @description: 删除文件函数
% */
function bitMap =deleteFile( bitMapRaw, fileToDelete)
    bitMap = bitMapRaw;
    blockNum = ceil(fileToDelete.len/2);
    startj = mod(fileToDelete.start - 1,size(bitMapRaw,2)) + 1;
    starti = (fileToDelete.start - startj)/size(bitMapRaw,2) + 1;
    cnt = 0;
    % 将该文件所对应的位在位示图中置 0
    for i = starti:size(bitMapRaw,1)
        for j = startj:size(bitMapRaw,2)
            bitMap(i,j) = 0;
            cnt = cnt +1;
            if(cnt >= blockNum)
                return;
            end
        end
    end
end
end
end

```

● main.m:

```

% /**
% *

```

```

% * @author:      胡文博
% * @email:       huwenbo@mail.dlut.edu.cn
% * @dateTime:    2017-06-11 22:57:22
% * @description:
% */

len = rand(50,1)*8 + 2;%生成 50 个随机长度作为文件长度
map = zeros(25,20);%位示图矩阵
storedFiles = [];
% 将随机生成的 50 个文件存储起来
for index = 1:size(len,1)
    [map,files] = saveFile(map,[num2str(index),'.txt'],len(index) );
    storedFiles = [storedFiles,files];
end
figure(1);%可视化当前位示图
imagesc(map);
title('Stored 1.txt ~ 50.txt ');
% 删除奇数.txt
for i = 1:size(storedFiles,2)/2
    map = deleteFile(map,storedFiles(1,i));
    storedFiles(i) = [];
end
figure(2);%可视化当前位示图
imagesc(map);
title('Deleted 1.txt, 3.txt, ... 49.txt ');
%创建题目（3）中的文件并存储起来
[map,files] = saveFile(map,'A.txt',7 );
storedFiles = [storedFiles,files];
[map,files] = saveFile(map,'B.txt',5 );
storedFiles = [storedFiles,files];
[map,files] = saveFile(map,'C.txt',2 );
storedFiles = [storedFiles,files];
[map,files] = saveFile(map,'D.txt',9 );
storedFiles = [storedFiles,files];
[map,files] = saveFile(map,'E.txt',3.5 );
storedFiles = [storedFiles,files];
figure(3);%可视化当前位示图
imagesc(map),colorbar;
for i = 1:size(storedFiles,2)
    disp(['name: ',storedFiles(1,i).name,',      start block:
',num2str(storedFiles(1,i).start),...
',      length: ',num2str(storedFiles(1,i).len),'k']);
end
title('Stored A.txt, B.txt, ... E.txt');

```

5.3 实验结果

实验结果如图所示，其中图 5.1 是存储了 1.txt, 2.txt, ... 50.txt 后的位示图，图 5.2 是删除奇数.txt（1.txt、3.txt、……、49.txt）文件后的位示图，图 5.3 是存储了新创建的 5 个文件（A.txt、B.txt、C.txt、D.txt、E.txt 大小为：7k、5k、2k、9k、3.5k）后的位示图。图 5.4 显示了各个文件的盘块存储状态。

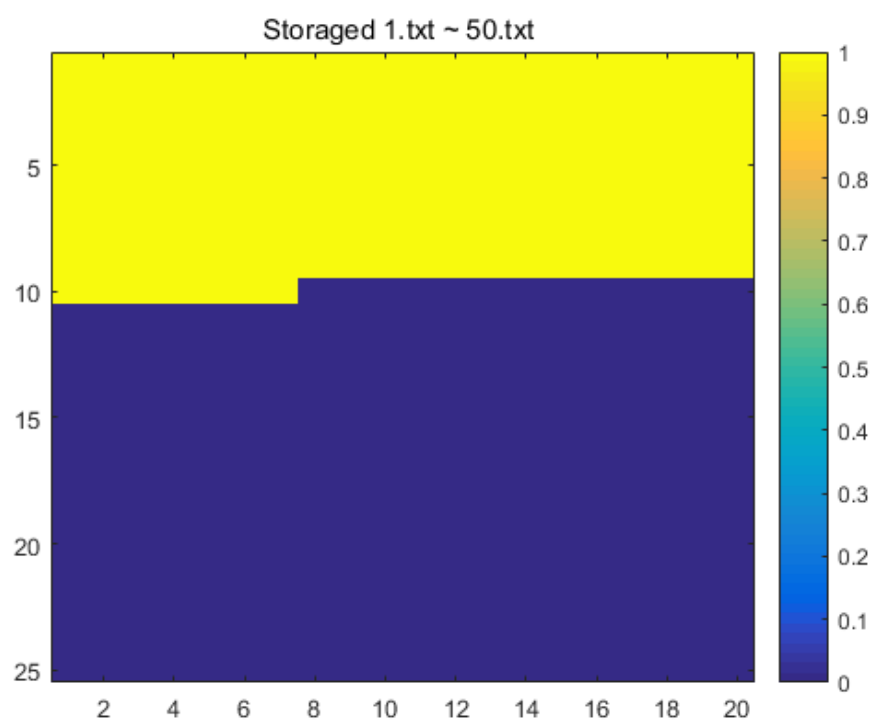


图 5.1

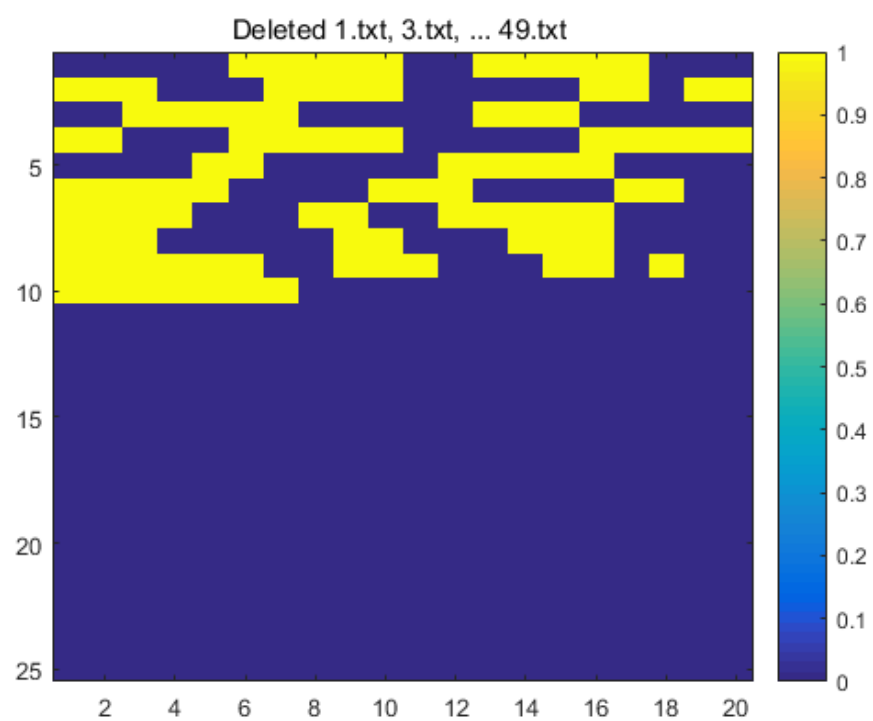


图 5.2

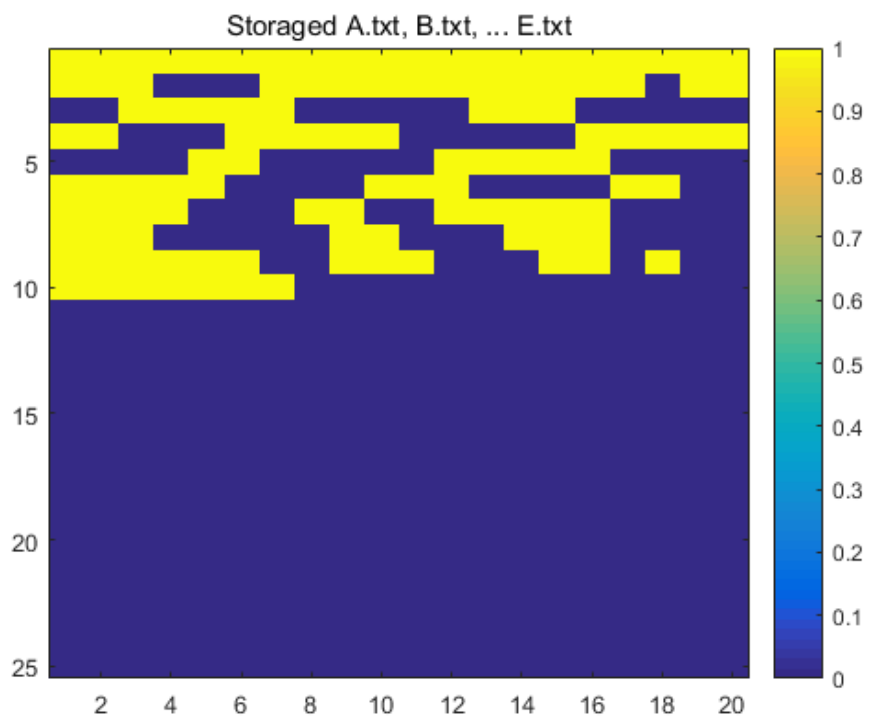
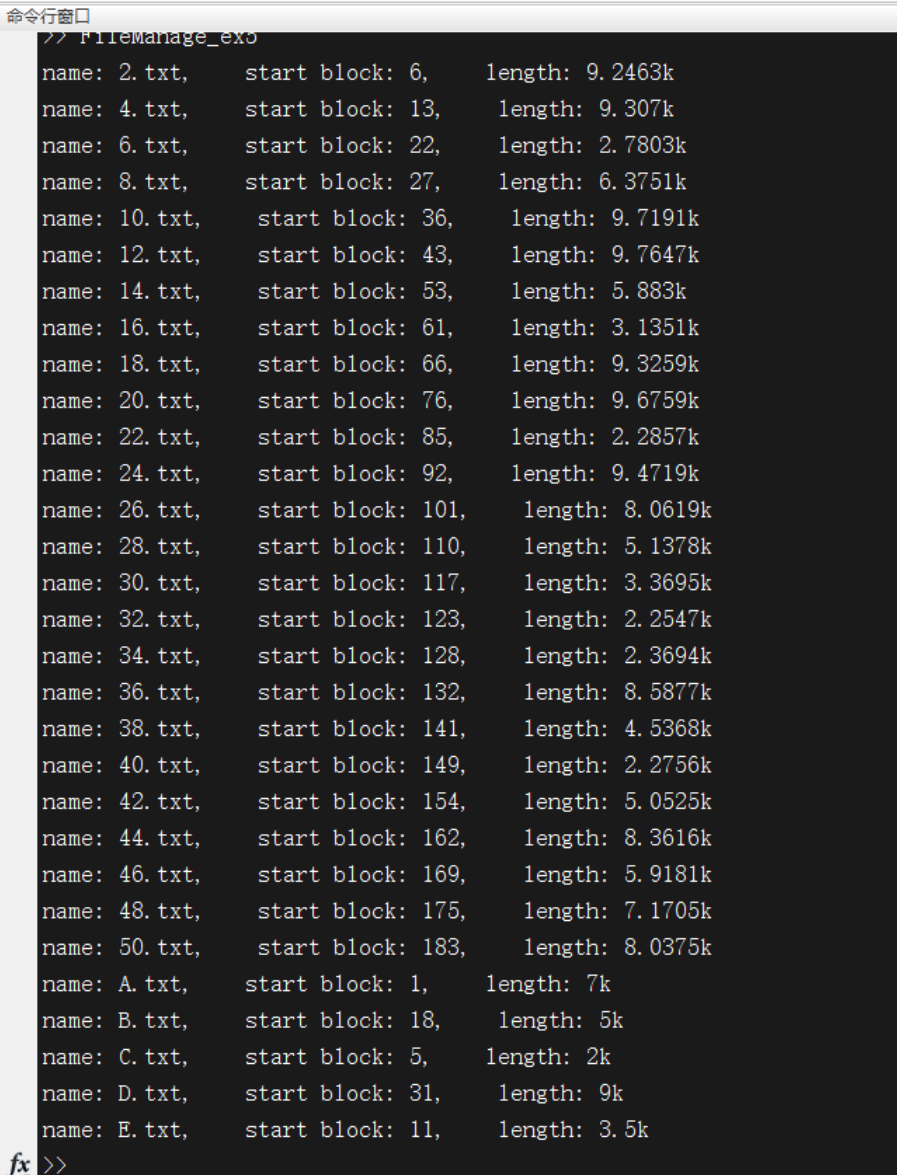


图 5.3



```
>> FileManage_exe
name: 2.txt,      start block: 6,      length: 9.2463k
name: 4.txt,      start block: 13,     length: 9.307k
name: 6.txt,      start block: 22,     length: 2.7803k
name: 8.txt,      start block: 27,     length: 6.3751k
name: 10.txt,     start block: 36,     length: 9.7191k
name: 12.txt,     start block: 43,     length: 9.7647k
name: 14.txt,     start block: 53,     length: 5.883k
name: 16.txt,     start block: 61,     length: 3.1351k
name: 18.txt,     start block: 66,     length: 9.3259k
name: 20.txt,     start block: 76,     length: 9.6759k
name: 22.txt,     start block: 85,     length: 2.2857k
name: 24.txt,     start block: 92,     length: 9.4719k
name: 26.txt,     start block: 101,    length: 8.0619k
name: 28.txt,     start block: 110,    length: 5.1378k
name: 30.txt,     start block: 117,    length: 3.3695k
name: 32.txt,     start block: 123,    length: 2.2547k
name: 34.txt,     start block: 128,    length: 2.3694k
name: 36.txt,     start block: 132,    length: 8.5877k
name: 38.txt,     start block: 141,    length: 4.5368k
name: 40.txt,     start block: 149,    length: 2.2756k
name: 42.txt,     start block: 154,    length: 5.0525k
name: 44.txt,     start block: 162,    length: 8.3616k
name: 46.txt,     start block: 169,    length: 5.9181k
name: 48.txt,     start block: 175,    length: 7.1705k
name: 50.txt,     start block: 183,    length: 8.0375k
name: A.txt,      start block: 1,      length: 7k
name: B.txt,      start block: 18,     length: 5k
name: C.txt,      start block: 5,      length: 2k
name: D.txt,      start block: 31,     length: 9k
name: E.txt,      start block: 11,     length: 3.5k
fx >>
```

图 5.4

收获与体会

经过本次操作系统实验，我对理论课所学习的各种算法有了更深层次的理解。以前学习各种算法只止步于表面原理，总感觉没有碰触到本质，在操作系统实验课上通过亲自编写程序模拟计算机操作系统中的各类算法，能学到很多理论课上没有领悟到的东西。比如在第一个实验中，在理论课上学习了进程的各种特性，但总感觉看不到摸不着，在实验中我们使用 linux 的系统函数 `fork()` 来克隆创建子进程，了解到克隆的子进程具

有和父进程完全相同的代码区和数据区，子进程的 PC 指针和父进程在创建他后一样，只能通过 `fork()` 函数的返回值来确定其为父进程还是子进程。

另外，在老师的指引下了解到了在计算机学科仅仅实现功能是不够的，还有考虑实现的效率，结果表现的直观性，输入的便捷性等等。在聆听了老师的教导后，我开始注重实验结果的可视化，让实验结果更加直观，让别人能一眼就看懂实验结果，比如在实验四的磁盘移臂调度算法中，我通过二维的折线图来表示调度的次序，并将不同算法的结果折线图放在一张图上，使人能一眼就看出两种调度算法的区别。

最后，很抱歉的是由于本报告采用黑白打印，故报告中的代码的高亮在纸质版上无法清晰地显示出来，如果老师想更清晰的阅读此报告的电子版，可以在本实验的 Github 项目主页上下载电子版，也可以在主页上看到动态的实验结果。

项目主页：https://github.com/crisb-DUT/OS_Experiment