# 实验四 处理器调度

24320162202888

牛晓彤

# 实验目的

* 理解进程调度的过程。
* 掌握各种进程调度算法的实现方法
* 通过实验比较各种进程调度算法的优劣。

# 实验内容

随机给出一个进程调度实例，如：

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

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SPN、最短剩余时间SRT、最高响应比优先HRRN进行调度各进程的完成时间、周转时间、响应比的值。

# 实验环境

* VisualBox + CentOS + GCC

# 实验中遇到的主要问题及其解决方式

问题: 每种算法在调用后没有正确的重新初始化进程状态数值,导致算法输出值与理论计算得出值不符

解决方式: 重写ClearProcInfor()/每次main()运行只调用一种算法

# 源代码

核心代码

void ScheduliAlgor::FCFS()

{

int runTime = 0;

for (int i = 0; i < length; i++)

{

if (proc[i].arrivedTime>runTime)

{

runTime = proc[i].arrivedTime;

}

runTime = runTime + proc[i].serverTime;

SetProInfor(i,runTime);

}

OutputInfor("FCFS");

ClearProcInfor();

}

void ScheduliAlgor::RR()

{

int runTime = 0;

int excutedProcNumbers = 0;

int currentProc = 0;

for (int i = 0; i < length; i++)

{

proInfor[i].remainTime = proc[i].serverTime;

}

while (excutedProcNumbers < length)

{

currentProc = FindNextExcuteProc(currentProc, runTime);

if (currentProc < 0)

{

++runTime;

}

else

{

--proInfor[currentProc].remainTime;

++runTime;

if(proInfor[currentProc].remainTime ==0 )

{

SetProInfor(currentProc,runTime);

excutedProcNumbers++;

}

}

}

OutputInfor("RR(q=1)");

ClearProcInfor();

}

void ScheduliAlgor::SPN()

{

int runTime = 0;

int excutedProcNumbers = 0;

int currentProc = 0;

while(excutedProcNumbers<length)

{

currentProc = FindTheShortestProc( runTime);

if(currentProc<0)

{

runTime++;

}

else

{

runTime = runTime + proc[currentProc].serverTime;

SetProInfor(currentProc,runTime);

++excutedProcNumbers;

}

}

OutputInfor("SPN");

ClearProcInfor();

}

void ScheduliAlgor::SRT()

{

int runTime = 0;

int excutedProcNumbers = 0;

int currentProc = 0;

for (int i = 0; i < length; i++)

{

proInfor[i].remainTime = proc[i].serverTime;

}

while(excutedProcNumbers < length)

{

currentProc = FindShortestRemain(runTime);

if(currentProc < 0)

{

++runTime;

}

else

{

++runTime;

--proInfor[currentProc].remainTime;

if(proInfor[currentProc].remainTime == 0)

{

SetProInfor(currentProc,runTime);

excutedProcNumbers++;

}

}

}

OutputInfor("SRT");

ClearProcInfor();

}

void ScheduliAlgor::HRRN( )

{

int currentProc = 0;

int runTime = 0;

int excutedProcNumbers = 0;

while (excutedProcNumbers < length)

{

currentProc = FindHightestRate(runTime);

if(currentProc < 0)

{

++runTime;

}

else

{

runTime = runTime + proc[currentProc].serverTime;

SetProInfor(currentProc, runTime);

++excutedProcNumbers;

}

}

OutputInfor("HRRN");

ClearProcInfor();

}

完整代码

scheduling.h

#include<iostream>

using namespace std;

class ScheduliAlgor

{

public:

ScheduliAlgor();

~ScheduliAlgor();

void InputProc();

void FCFS();

void RR();

void SPN();

void SRT();

void HRRN();

private:

void OutputInfor( char \* name);

int FindTheShortestProc( int runTime);

int FindHightestRate (int runTime);

int FindShortestRemain(int runTime);

int FindNextExcuteProc( int currentProc, int runTime);

void SetProInfor(int currentProc , int runTime);

void ClearProcInfor();

struct Proc

{

char ID;

unsigned int arrivedTime;

unsigned int serverTime;

};

struct ProcInfor

{

unsigned int finishTime;

unsigned int turnaroundTime;

double rate;

bool isExcuted;

bool isFirst;

int remainTime;

};

int length;

struct Proc \*proc;

struct ProcInfor \*proInfor;

void put();

};

scheduling.cpp

#include"scheduling.h"

ScheduliAlgor::ScheduliAlgor()

{

proc = NULL;

proInfor = NULL;

length = 0;

}

ScheduliAlgor::~ScheduliAlgor()

{

delete [] proc;

proc = NULL;

delete [] proInfor;

proInfor = NULL;

}

void ScheduliAlgor::InputProc()

{

cout<<"Input the numbers of proc:";

cin>>length;

if(length<1)

{

cout<<"Error"<<endl;

return ;

}

proc = new struct Proc[length];

if(proc == NULL)

{

cout<<"Error"<<endl;

return ;

}

proInfor = new struct ProcInfor[length];

if (proInfor == NULL)

{

cout<<"Error"<<endl;

return ;

}

ClearProcInfor();

cout<<"Input arrive time and sever time:"<<endl;

for (int i = 0; i < length; i++)

{

cin>>proc[i].arrivedTime>>proc[i].serverTime;

proc[i].ID = 'A'+i;

if (proc[i].serverTime == 0)

{

cout<<"Error"<<endl;

return ;

}

}

}

void ScheduliAlgor::FCFS()

{

int runTime = 0;

for (int i = 0; i < length; i++)

{

if (proc[i].arrivedTime>runTime)

{

runTime = proc[i].arrivedTime;

}

runTime = runTime + proc[i].serverTime;

SetProInfor(i,runTime);

}

OutputInfor("FCFS");

ClearProcInfor();

}

void ScheduliAlgor::RR()

{

int runTime = 0;

int excutedProcNumbers = 0;

int currentProc = 0;

for (int i = 0; i < length; i++)

{

proInfor[i].remainTime = proc[i].serverTime;

}

while (excutedProcNumbers < length)

{

currentProc = FindNextExcuteProc(currentProc, runTime);

if (currentProc < 0)

{

++runTime;

}

else

{

--proInfor[currentProc].remainTime;

++runTime;

if(proInfor[currentProc].remainTime ==0 )

{

SetProInfor(currentProc,runTime);

excutedProcNumbers++;

}

}

}

OutputInfor("RR(q=1)");

ClearProcInfor();

}

int ScheduliAlgor::FindNextExcuteProc( int currentProc, int runTime)

{

if (proc[0].arrivedTime > runTime)

{

return -1;

}

int tMark = currentProc;

int mark = --currentProc;

if(mark<0)

mark+= length;

int excuteTime = 0;

bool isFind = false;

bool isFirstOne = true;

while (excuteTime < length)

{

if(proInfor[mark].isExcuted)

{ }

else

{

if (proc[mark].arrivedTime<=runTime)

{

if (isFirstOne)

{

isFirstOne = !isFirstOne;

isFind = true;

currentProc = mark;

}

else

{

if (tMark == mark)

{ }

else if(( proc[mark].arrivedTime<runTime)&&

(proc[currentProc].arrivedTime==runTime))

{

currentProc = mark;

}

}

}

}

--mark;

if(mark<0)

mark+=length;

++excuteTime;

}

if(isFind)

return currentProc;

return -1;

}

void ScheduliAlgor::SPN()

{

int runTime = 0;

int excutedProcNumbers = 0;

int currentProc = 0;

while(excutedProcNumbers<length)

{

currentProc = FindTheShortestProc( runTime);

if(currentProc<0)

{

runTime++;

}

else

{

runTime = runTime + proc[currentProc].serverTime;

SetProInfor(currentProc,runTime);

++excutedProcNumbers;

}

}

OutputInfor("SPN");

ClearProcInfor();

}

int ScheduliAlgor::FindTheShortestProc( int runTime)

{

int mark = -1;

bool isFirstOne = true;

for (int i = 0; i < length; i++)

{

if (!proInfor[i].isExcuted)

{

if (proc[i].arrivedTime<=runTime)

{

if(isFirstOne)

{

mark = i;

isFirstOne = !isFirstOne;

}

else

{

if(proc[i].serverTime<proc[mark].serverTime)

mark = i;

}

}

}

}

return mark;

}

void ScheduliAlgor::SRT()

{

int runTime = 0;

int excutedProcNumbers = 0;

int currentProc = 0;

for (int i = 0; i < length; i++)

{

proInfor[i].remainTime = proc[i].serverTime;

}

while(excutedProcNumbers < length)

{

currentProc = FindShortestRemain(runTime);

if(currentProc < 0)

{

++runTime;

}

else

{

++runTime;

--proInfor[currentProc].remainTime;

if(proInfor[currentProc].remainTime == 0)

{

SetProInfor(currentProc,runTime);

excutedProcNumbers++;

}

}

}

OutputInfor("SRT");

ClearProcInfor();

}

int ScheduliAlgor::FindShortestRemain(int runTime)

{

bool isFirstOne = true;

int shortestRemainTime = 0;

int mark = -1;

for (int i = 0; i < length; i++)

{

if (!proInfor[i].isExcuted)

{

if(proc[i].arrivedTime<=runTime)

{

if(isFirstOne)

{

shortestRemainTime = proInfor[i].remainTime;

mark = i;

isFirstOne = !isFirstOne;

}

else

{

int tempTime = proInfor[i].remainTime;

if(shortestRemainTime > tempTime)

{

shortestRemainTime = tempTime;

mark = i;

}

}

}

}

}

return mark;

}

void ScheduliAlgor::HRRN( )

{

int currentProc = 0;

int runTime = 0;

int excutedProcNumbers = 0;

while (excutedProcNumbers < length)

{

currentProc = FindHightestRate(runTime);

if(currentProc < 0)

{

++runTime;

}

else

{

runTime = runTime + proc[currentProc].serverTime;

SetProInfor(currentProc, runTime);

++excutedProcNumbers;

}

}

OutputInfor("HRRN");

ClearProcInfor();

}

int ScheduliAlgor::FindHightestRate(int runTime)

{

int mark = -1;

bool isFirstOne = true;

double rate=0;

for (int i = 0; i < length; i++)

{

if(!proInfor[i].isExcuted)

{

if(proc[i].arrivedTime <= runTime)

{

if(isFirstOne)

{

mark = i;

rate = (double)(runTime + proc[i].serverTime - proc[i].arrivedTime)/(double)proc[i].serverTime;

isFirstOne = !isFirstOne;

}

else

{

double tempRate = (double)(runTime + proc[i].serverTime - proc[i].arrivedTime)/(double)proc[i].serverTime;

if (tempRate > rate)

{

rate = tempRate;

mark = i;

}

}

}

}

}

return mark;

}

void ScheduliAlgor::ClearProcInfor()

{

for (int i = 0; i < length; i++)

{

proInfor[i].finishTime = 0;

proInfor[i].turnaroundTime = 0;

proInfor[i].rate = 0;

proInfor[i].turnaroundTime = 0;

proInfor[i].remainTime = 0;

proInfor[i].isExcuted = false;

proInfor[i].isFirst = true;

}

}

void ScheduliAlgor::SetProInfor(int currentProc , int runTime)

{

proInfor[currentProc].finishTime = runTime;

proInfor[currentProc].isExcuted = true;

proInfor[currentProc].turnaroundTime = proInfor[currentProc].finishTime - proc[currentProc].arrivedTime;

proInfor[currentProc].rate = (double)proInfor[currentProc].turnaroundTime/(double)proc[currentProc].serverTime;

}

void ScheduliAlgor::OutputInfor(char \*name)

{

double totalTurnaroundTime = 0;

double totalRate = 0;

cout<<"Scheduling discipline is "<<name<<""<<endl;

cout<<" finish "<<"turnaround "<<"rate"<<endl;

for (int i = 0; i < length; i++)

{

cout<<proc[i].ID<<"\t"<<proInfor[i].finishTime<<"\t"<<proInfor[i].turnaroundTime<<"\t"<<proInfor[i].rate<<endl;

}

}

main.cpp

#include<iostream>

#include"scheduling.h"

using namespace std;

int main()

{

ScheduliAlgor \*scheduliAlgor = new ScheduliAlgor;

scheduliAlgor->InputProc();

scheduliAlgor->FCFS();

//scheduliAlgor->RR();

//scheduliAlgor->SPN();

//scheduliAlgor->SRT();

//scheduliAlgor->HRRN();

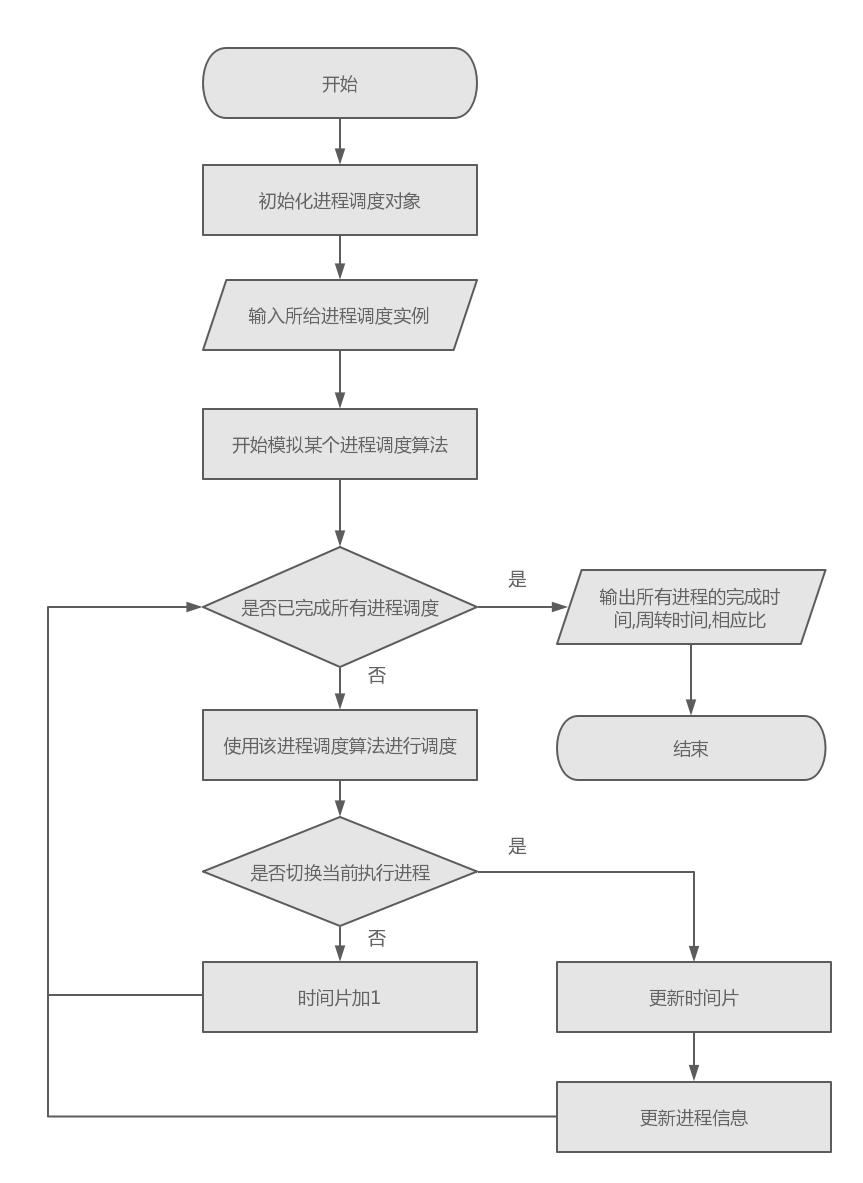
getchar();

getchar();

return 0;

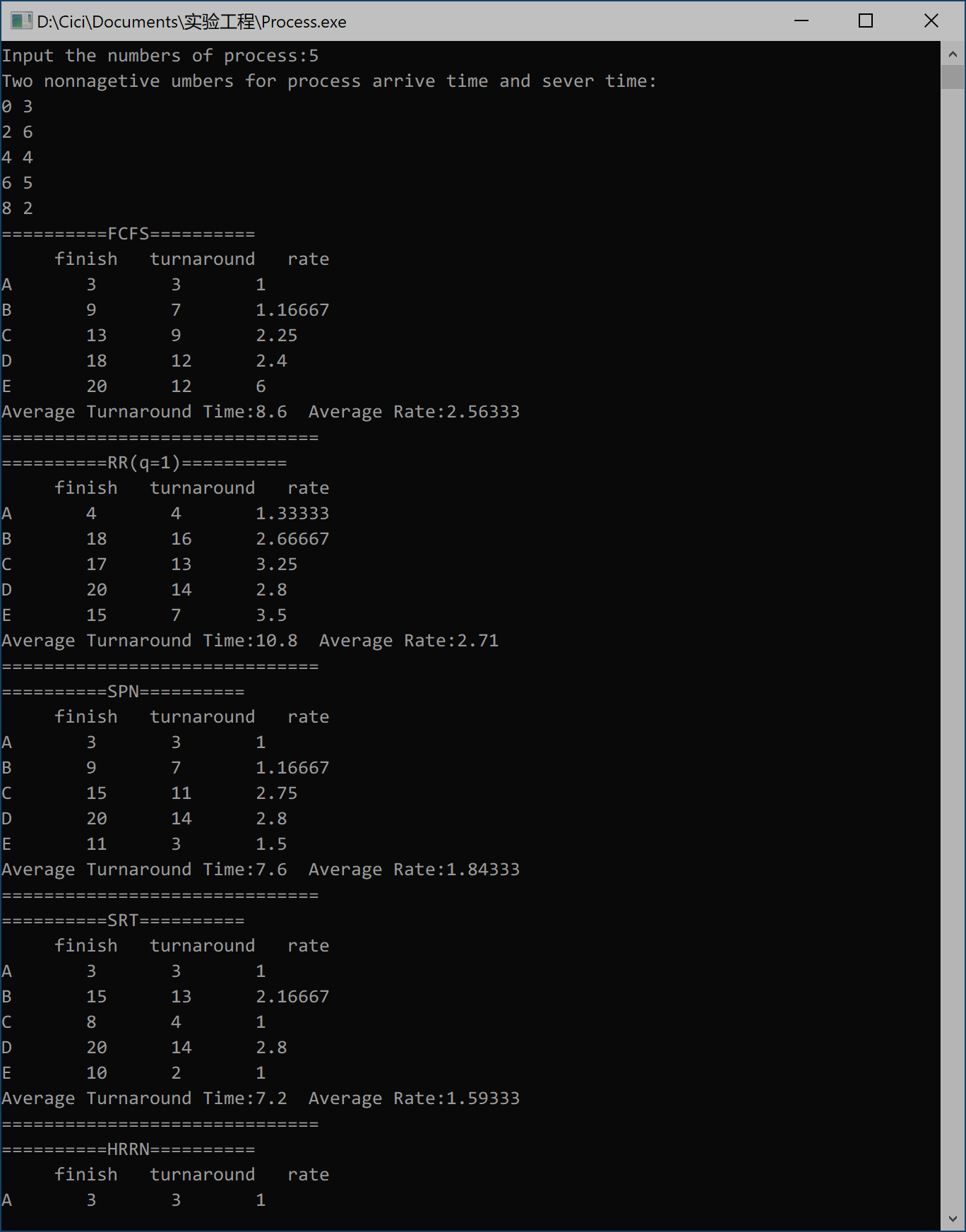
}

# 程序流程图



# 实验总结

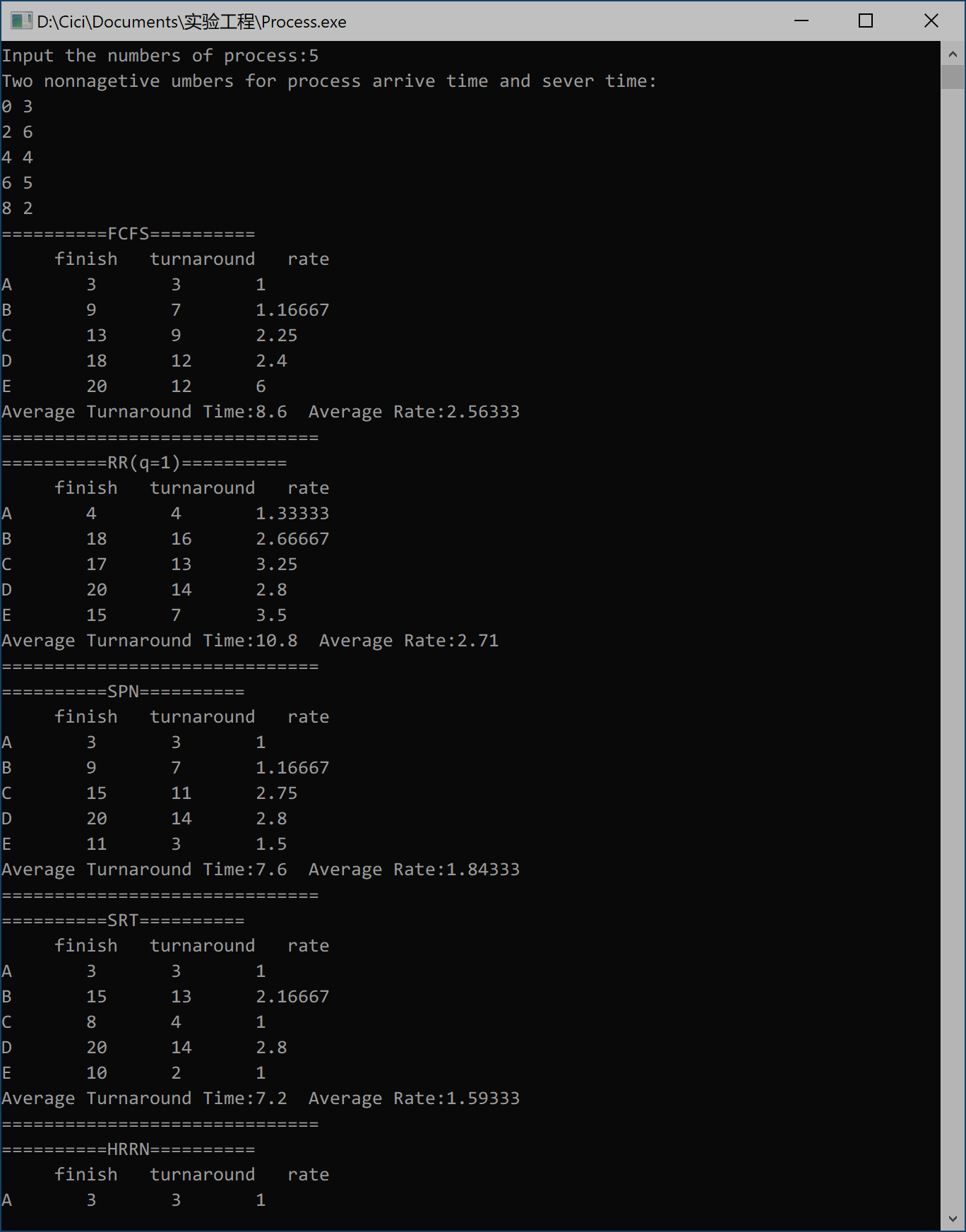
输入5个进程到达时间和服务时间,



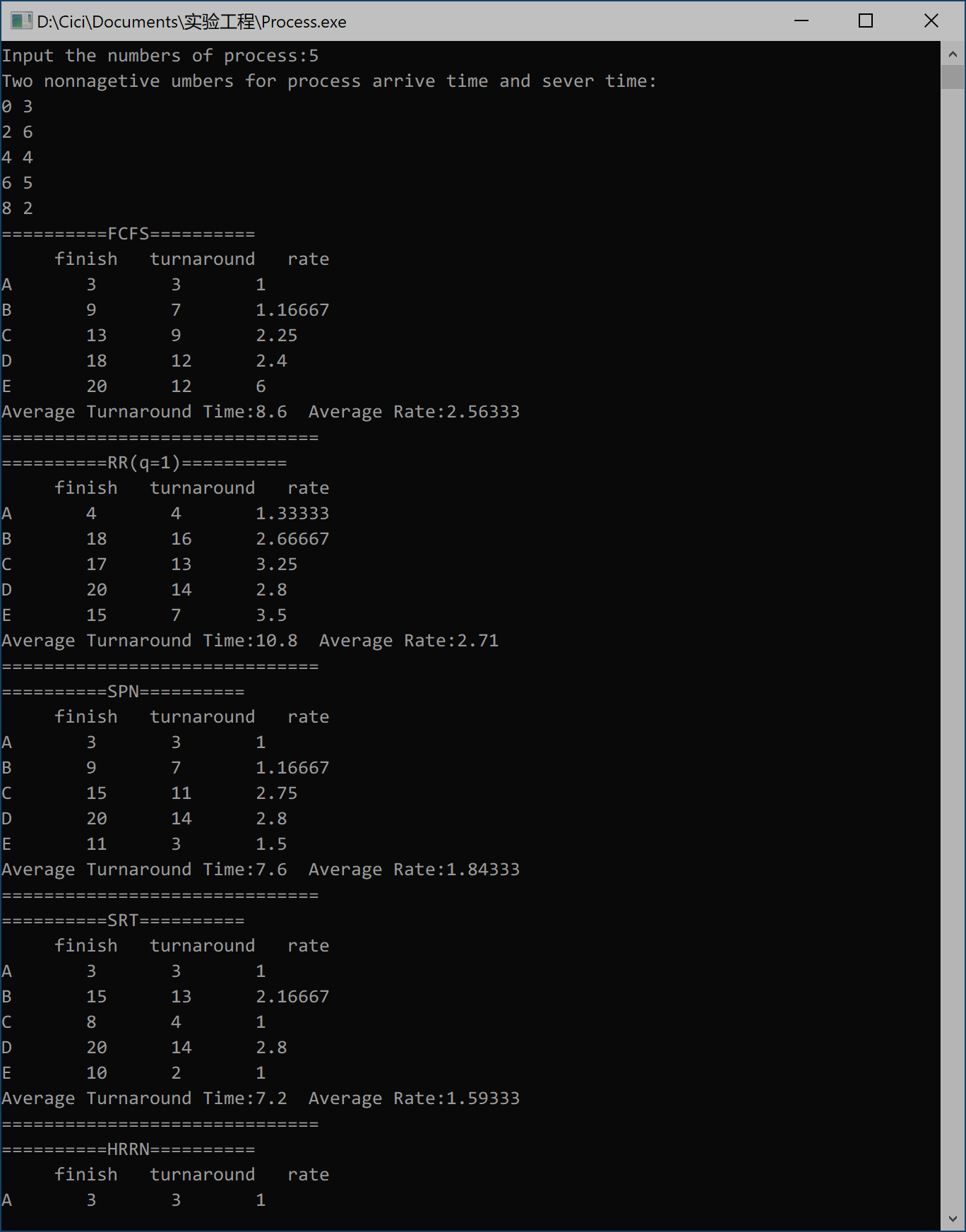
分别调动服务算法FCFS/RR(q=1)/SPN/SPT/HRRN

程序输出如下:

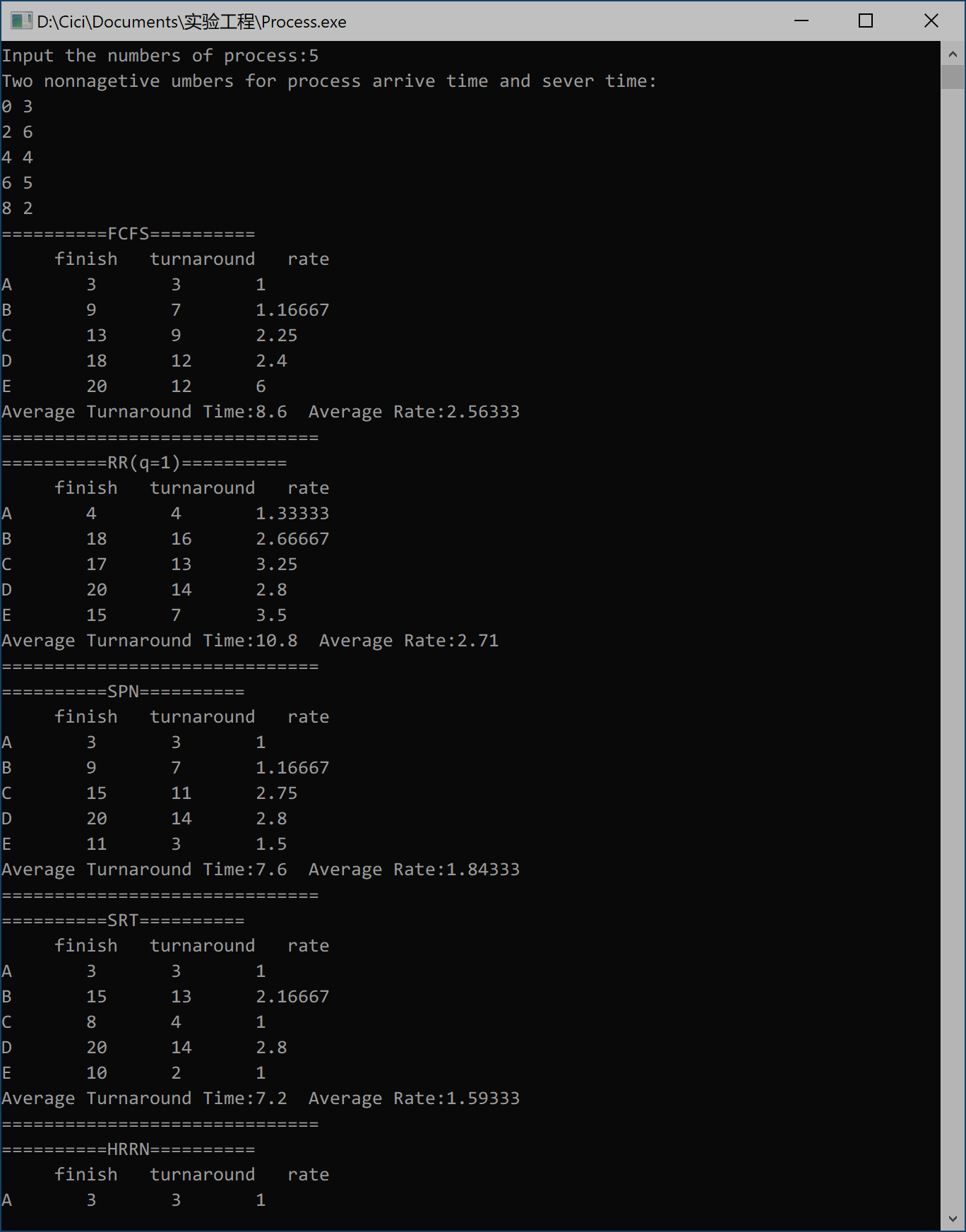
FCFS：



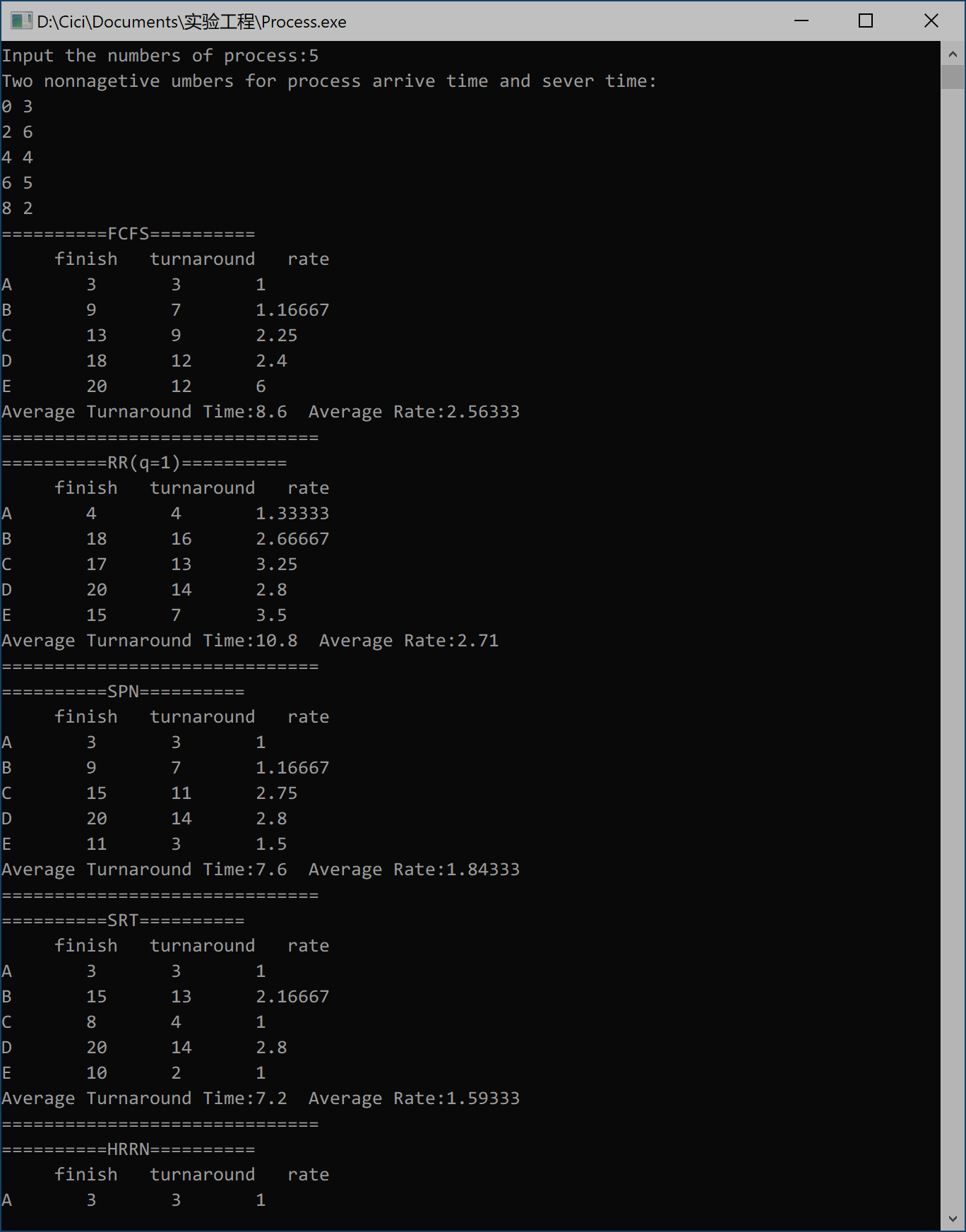
RR:



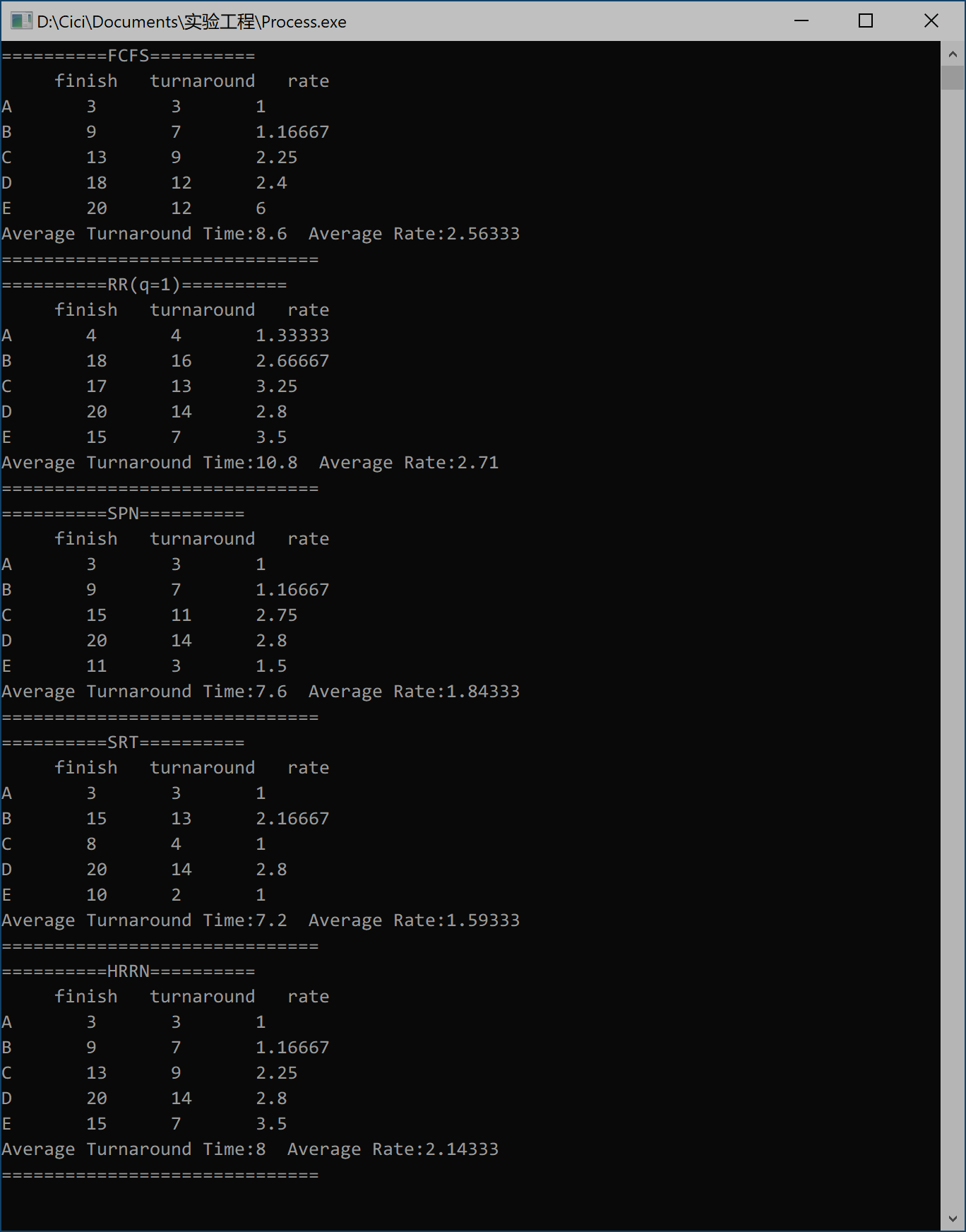
SPN



SRT



HRRN



* 1. FCFS:这种策略是当当前正在运行的进程停止执行时，选择在就绪队列中存在时间最长的进程运行。这种策略执行长进程比执行短进程更好。
  2. 轮转:这种策略是以一个周期性间隔产生时钟中断，当中断发生时，当前正在运行的进程被置于就绪队列中，然后基于FCFS策略选择下一个就绪作业运行，目的是为了减少在FCFS策略下短作业的不利情况。
  3. SPN最短进程优先:这种策略是下一次选择所需处理时间最短的进程。是非抢占策略，目的也是为减少FCFS策略对长进程的偏向。
  4. SRT最短剩余时间:这种策略下调度器总是选择预期剩余时间最短的进程。是抢占策略。
  5. HRRN最高响应比优先:这种策略是当当前进程完成或被阻塞时，选择响应比R最大的就绪进程，R=(w+s)/s 。这样长进程被饿死的可能性下降。